

CI/CD WITH KUBERNETES

The New Stack

CI/CD with Kubernetes

Alex Williams, Founder & Editor-in-Chief

Core Team:

Bailey Math, AV Engineer

Benjamin Ball, Marketing Director

Gabriel H. Dinh, Executive Producer

Judy Williams, Copy Editor

Kiran Oliver, Podcast Producer

Lawrence Hecht, Research Director

Libby Clark, Editorial Director

Norris Deajon, AV Engineer

© 2018 The New Stack. All rights reserved.

20180609

TABLE OF CONTENTS

Introduction 4

Sponsors 7

Contributors 8

CI/CD WITH KUBERNETES

DevOps Patterns 9

KubeCon + CloudNativeCon: The Best CI/CD Tool For Kubernetes Doesn't Exist 39

Cloud-Native Application Patterns 40

Aqua Security: Improve Security with Automated Image Scanning Through CI/CD.... 61

Continuous Delivery with Spinnaker..... 62

Google Cloud: A New Approach to DevOps With Spinnaker on Kubernetes 88

Monitoring in the Cloud-Native Era 89

Closing..... 115

Disclosure 117

INTRODUCTION

Kubernetes is the cloud orchestrator of choice. Its core is like a hive: orchestrating containers, scheduling, serving as a declarative infrastructure on self-healing clusters. With its capabilities growing at such a pace, Kubernetes' ability to scale forces questions about how an organization manages its own teams and adopts DevOps practices. Historically, continuous integration has offered a way for DevOps teams to get applications into production, but continuous delivery is now a matter of increasing importance. How to achieve continuous delivery will largely depend on the use of distributed architectures that manage services on sophisticated and fast infrastructure that use compute, networking and storage for continuous, on-demand services. Developers will consume services as voraciously as they can to achieve the most out of them. They will try new approaches for development, deployment and, increasingly, the management of microservices and their overall health and behavior.

Kubernetes is similar to other large-scope, cloud software projects that are so complex that their value is only determined when they are put into practice. The container orchestration technology is increasingly being used as a platform for application deployment defined by the combined forces of DevOps, continuous delivery and observability. When employed together, these three forces deliver applications faster, more efficiently and closer to what customers want and demand. Teams start by building applications as a set of microservices in a container-based, cloud-native architecture. But DevOps practices are what truly transform the application architectures of an organization; they are the basis for all of the patterns and practices that make applications run on Kubernetes. And DevOps transformation only comes with aligning an organization's values with the ways it develops application architectures.

In this newly optimized means to cloud-native transformation, Kubernetes is the enabler — it's not a complete solution. Your organization must implement the tools and practices best suited to your own business needs and structure in order to realize the full promise of this open source platform. The [Kubernetes project documentation](#) itself says so:

Kubernetes “does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.”

This ebook, the third and final in The New Stack's Kubernetes ecosystem series, lays the foundation for understanding and building your team's practices and pipelines for delivering — and continuously improving — applications on Kubernetes. How is that done? It's not a set of rules. It's a set of practices that flow into the organization and affect how application architectures are developed. This is DevOps, and its currents are now deep inside organizations with modern application architectures, manifested through continuous delivery.

Section Summaries

- **Section 1: DevOps Patterns** by Rob Scott of ReactiveOps, explores the history of DevOps, how it is affecting cloud-native architectures and how Kubernetes is again transforming DevOps. This section traces the history of Docker and container packaging to the emergence of Kubernetes and how it is affecting application development and deployment.
- **Section 2: Cloud-Native Application Patterns** is written by Janakiram MSV, principal analyst at Janakiram & Associates. It reviews how Kubernetes manages resource allocation automatically, according

to policies set out by DevOps teams. It details key cloud-native attributes, and maps workload types to Kubernetes primitives.

- **Section 3: Continuous Delivery with Spinnaker** by Craig Martin, senior vice president of engineering at Kenzan, analyzes how continuous delivery with cloud-native technologies requires deeper understanding of DevOps practices and how that affects the way organizations deploy and manage microservices. Spinnaker is given special attention as an emerging CD tool that is itself a cloud-native, microservices-based application.
- **Section 4: Monitoring in the Cloud-Native Era** by a team of engineers from Container Solutions, explains how the increasing complexity of microservices is putting greater emphasis on the need for combining traditional monitoring practices to gain better observability. They define observability for scaled-out applications running on containers in an orchestrated environment, with a specific focus on Prometheus as an emerging management tool.

While the book ends with a focus on observability, it's increasingly clear that cloud-native monitoring is not an endpoint in the development life cycle of an application. It is, instead, the process of granular data collection and analysis that defines patterns and informs developers and operations teams from start to finish, in a continual cycle of improvement and delivery. Similarly, this book is intended as a reference throughout the planning, development, release, manage and improvement cycle.

SPONSORS

We are grateful for the support of our ebook foundation sponsor:



And our sponsors for this ebook:



CONTRIBUTORS



[Rob Scott](#) works out of his home in Chattanooga as a Site Reliability Engineer for ReactiveOps. He helps build and maintain highly scalable, Kubernetes-based infrastructure for multiple clients. He's been working with Kubernetes since 2016, contributing to the official documentation along the way. When he's not building world-class infrastructure, Rob likes spending time with his family, exploring the outdoors, and giving talks on all things Kubernetes.



[Janakiram MSV](#) is the Principal Analyst at Janakiram & Associates and an adjunct faculty member at the International Institute of Information Technology. He is also a Google Qualified Cloud Developer; an Amazon Certified Solution Architect, Developer, and SysOps Administrator; a Microsoft Certified Azure Professional; and one of the first Certified Kubernetes Administrators and Application Developers. His previous experience includes Microsoft, AWS, Gigaom Research, and Alcatel-Lucent.



[Craig Martin](#) is Kenzan's senior vice president of engineering, where he helps to lead the technical direction of the company ensuring that new and emerging technologies are explored and adopted into the strategic vision. Recently, Craig has been focusing on helping companies make a digital transformation by building large-scale microservices applications. Prior to Kenzan, Craig was director of engineering at Flatiron Solutions.

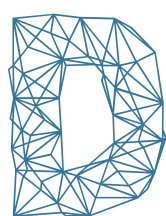


Container Solutions

Ian Crosby, Maarten Hoogendoorn, Thijs Schnitger and Etienne Tremel are engineers and experts in application deployment on Kubernetes for [Container Solutions](#), a consulting organization that provides support for clients who are doing cloud migrations.

DEVOPS PATTERNS

by **ROB SCOTT**



DevOps practices run deep in modern application architectures. DevOps practices have helped create a space for developers and engineers to build new ways to optimize resources and scale out application architectures through continuous delivery practices. Cloud-native technologies use the efficiency of containers to make microservices architectures that are more useful and adaptive than composed or monolithic environments. Organizations are turning to DevOps principles as they build cloud-native, microservices-based applications. The combination of DevOps and cloud-native architectures is helping organizations meet their business objectives by fostering a streamlined, lean product development process that can adapt quickly to market changes.

Cloud-native applications are based on a set of loosely coupled components, or microservices, that run for the most part on containers, and are managed with orchestration engines such as Kubernetes.

However, they are also beginning to run as a set of discrete functions in serverless architectures. Services or functions are defined by developer and engineering teams, then continuously built, rebuilt and improved by increasingly cross-functional teams. Operations are now less focused on

the infrastructure and more on the applications that run light workloads. The combined effect is a shaping of automated processes that yield better efficiencies.

In fact, some would argue that an application isn't truly cloud native unless it has DevOps practices behind it, as cloud-native architectures are built for web-scale computing. DevOps professionals are required to build, deploy and manage declarative infrastructure that is secure, resilient and high performing. Delivering these requirements just isn't feasible with a traditional siloed approach.

As the de facto platform for cloud-native applications, Kubernetes not only lies at the center of this transformation, but also enables it by abstracting away the details of the underlying compute, storage and networking resources. The open source software provides a consistent platform on which containerized applications can run, regardless of their individual runtime requirements. With Kubernetes, your servers can be dumb — they don't care what they're running. Instead of running a specific application on a specific server, multiple applications can be distributed across the same set of servers. Kubernetes simplifies application updates, enabling teams to deliver applications and features into users' hands quickly.

In order to find success with DevOps, however, a business must be intentional in its decision to build a cloud-native application. The organizational transformation required to put DevOps into practice will happen only if a business team is willing to invest in DevOps practices — transformation comes with the alignment of the product team in the development of the application. Together, these teams create the environment needed to continually refine technical development into lean, streamlined workflows that reflect continuous delivery processes built on DevOps principles.

For organizations using container orchestration technologies, product direction is defined by developing a microservices architecture. This is possible only when the organization understands how DevOps and continuous development processes enable the creation of applications that end users truly find useful.

Therein lies the challenge: You must make sure your organization is prepared to transform the way all members of the product team work. Ultimately, DevOps is a story about why you want to do streamlined, lean product development in the first place — the same reason that you're moving to a microservices architecture on top of Kubernetes.

Our author for this chapter is Rob Scott, a site reliability engineer at [ReactiveOps](#). Scott is an expert in DevOps practices, applying techniques from his learnings to help customers run services that can scale on Kubernetes architectures. His expertise in building scaled-out architectures stems from years of experience that has given him witness to:

- How containers brought developers and operators together into the field of DevOps.
- The role a container orchestration tool like Kubernetes plays in the container ecosystem.
- How Kubernetes resulted in a revolutionary transformation of the entire DevOps ecosystem — which is ultimately transforming businesses.

Traditional DevOps patterns before containers required different processes and workflows. Container technologies are built with a DevOps perspective. The abstraction containers offer is having an effect on how we view DevOps, as traditional architecture development changes with the advent of microservices. It means following best practices for running

containers on Kubernetes, and the extension of DevOps into GitOps and SecOps practices.

The Evolution of DevOps and CI/CD Patterns

A Brief History of DevOps

DevOps was born roughly 10 years ago, though organizations have shown considerably more interest in recent years. Half of organizations surveyed implemented DevOps practices in 2017, [according to Forrester Research](#), which has declared 2018 “The Year of Enterprise DevOps.” Although DevOps is a broad concept, the underlying idea involves development and operations teams working more closely together.

Traditionally, the speed with which software was developed and deployed didn’t allow a lot of time for collaboration between engineers and operations staff, who worked on separate teams. Many organizations had embraced lean product development practices and were under constant pressure to release software quickly. Developers would build out their applications, and the operations team would deploy them. Any conflict between the two teams resulted from a core disconnect — the operations team was unfamiliar with the applications being deployed, and the development team was unfamiliar with how the applications were being deployed.

As a result, application developers sometimes found that their platform wasn’t configured in a way that best met their needs. And because the operations team didn’t always understand software and feature requirements, at times they over-provisioned or under-provisioned resources. What happened next is no mystery: Operations teams were held responsible for engineering decisions that negatively impacted application performance and reliability. Worse, poor outcomes impacted the organization’s bottom line.

A key concept of DevOps involved bringing these teams together. As development and operations teams started to collaborate more frequently, it became clear that automation would speed up deployments and reduce operational risk. With these teams working closely together, some powerful DevOps tooling was built. These tools automated what had been repetitive, manual and error-prone processes with code.

Eventually these development and operations teams started to form their own “DevOps” teams that combined engineers from development and operations backgrounds. In these new teams, operations engineers gained development experience, and developers gained exposure to the behind-the-scenes ways that applications run. As this new specialization continues to evolve, next-generation DevOps tooling is being designed and built that will continue to transform the industry. Increased collaboration is still necessary for improved efficiencies and business outcomes, but further advantages of DevOps adoption are emerging. Declarative environments made possible by cloud-native architectures and managed through continuous delivery pipelines have lessened reliance on collaboration and shifted the focus toward application programming interface (API) calls and automation.

The Evolution of CI/CD Workflows

There are numerous models for developing iterative software, as well as an infinite number of continuous integration/continuous delivery (CI/CD) practices. While CI/CD processes aren’t new to the scene, they were more complex at the start. Now, continuous delivery has come to the fore as the next frontier for improved efficiencies as more organizations migrate to microservices and container-based architectures. A whole new set of tools and best practices are emerging that allow for increasingly automated and precise deployments, using strategies such as red/black deployments and automated canary analysis (ACA). [Chapter 3](#) has more detail.

Before the idea of immutable infrastructure gained popularity, servers were generally highly specialized and difficult to replace. Each server would have a specific purpose and would have been manually tuned to achieve that purpose. Tools like [Chef](#) and [Puppet](#) popularized the notion of writing reproducible code that could be used to build and tune these servers. Servers were still changing frequently, but now code was committed into version control. Changes to servers became simpler to track and recreate. These tools also started to simplify integration with CI/CD workflows. They enabled a standard way to pull in new code and restart an application across all servers. Of course, there was always a chance that the latest application could break, resulting in a situation that could be difficult to recover from quickly.

With that in mind, the industry started to move toward a pattern that avoided making changes to existing servers: immutable infrastructure. Virtual machines combined with cloud infrastructure to dramatically simplify creating new servers for each application update. In this workflow, a CI/CD pipeline would create machine images that included the application, dependencies and base operating system (OS). These machine images could then be used to create identical, immutable servers to run the application. They could also be tested in a quality assurance (QA) environment before being deployed to production.

The ability to test every bit of the image before it reached production resulted in an incredible improvement in reliability for QA teams.

Unfortunately, the process of creating new machine images and then running a whole new set of servers with them was also rather slow.

It was around this time that Docker started to gain popularity. Based on Linux kernel features, cgroups and namespaces, Docker is an open source project that automates the development, deployment and running of applications inside isolated containers. Docker offered a lot of the same

advantages as machine images, but it did so with a much more lightweight image format. Instead of including the whole base operating system, Docker images simply included the application and its dependencies. This process still provided the reliability advantages described earlier, but came with some substantial improvements in speed. Docker images were much faster to build, faster to pull in and faster to start up. Instead of creating new servers for each new deployment, new Docker containers were created that could run on the same servers.

With the lightweight approach Docker provided, CI/CD workflows really started to take off. For example, each new commit to your Git repository could have a corresponding Docker image built. Each Git commit could trigger a multi-step, customizable build process that includes vulnerability scanning for container images. Cached images could then be used for subsequent builds, speeding up the build process in future iterations. One of the most recent improvements in these workflows has come with container orchestration tools like Kubernetes. These tools have dramatically simplified deployment of application updates with containers. In addition, they have had transformative effects on resource utilization. Whereas before you might have run a single application on a server, with container orchestration multiple containers with vastly different workloads can run on the same server. With Kubernetes, CI/CD is undergoing yet another evolution that has tremendous implications for the business efficiencies gained through DevOps.

Modern DevOps Practices

Docker was the first container technology to gain broad popularity, though alternatives exist and are standardized by the Open Container Initiative (OCI). Containers allow developers to bundle up an application with all of the dependencies it needs to run and package and ship it in a

single package. Before, each server would need to have all the OS-level dependencies to run a Ruby or Java application. The container changes that. It's a thin wrapper — single package — containing everything you need to run an application. Let's explore how modern DevOps practices reflect the core value of containers.

Containers Bring Portability

Docker is both a daemon — a process running in the background — and a client command. It's like a virtual machine, but it's different in important ways. First, there's less duplication. With each extra virtual machine (VM) you run, you duplicate the virtualization of central processing units (CPUs) and memory and quickly run out of local resources. Docker is great at setting up a local development environment because it easily adds the running process without duplicating the virtualized resource. Second, it's more modular. Docker makes it easy to run multiple versions or instances of the same program without configuration headaches and port collisions.

Thus, instead of a single VM or multiple VMs, you can link each individual application and supporting service into a single unit and horizontally scale individual services without the overhead of a VM. And it does it all with a single descriptive Dockerfile syntax, improving the development experience, speeding software delivery and boosting performance. And because Docker is based on open source technology, anyone can contribute to its development to build out features that aren't yet available.

With Docker, developers can focus on writing code without worrying about the system on which their code will run. Applications become truly portable. You can repeatedly run your application on any other machine running Docker with confidence. For operations staff, Docker is lightweight, easily allowing the running and management of applications

with different requirements side by side in isolated containers. This flexibility can increase resource utilization per server and may reduce the number of systems needed due to lower overhead, which in turn reduces cost.

Containers Further Blur the Lines Between Operations and Development

Containers represent a significant shift in the traditional relationship between development and operations teams. Specifications for building a container have become remarkably straightforward to write, and this has increasingly led to development teams writing these specifications. As a result, development and operations teams work even more closely together to deploy these containers.

The popularity of containers has led to significant improvements for CI/CD pipelines. In many cases, these pipelines can be configured with some simple YAML files. This pipeline configuration generally also lives in the same repository as the application code and container specification. This is a big change from the traditional approach in which code to build and deploy applications is stored in a separate repository and entirely managed by operations teams.

With this move to a simplified build and deployment configuration living alongside application code, developers are becoming increasingly involved in processes that were previously managed entirely by operations teams.

Initial Challenges with Containers

Though containers are now widely adopted by most organizations, there have historically been three basic challenges that prevented organizations from making the switch. First, it takes a mindshift to translate a current development solution into a containerized development solution. For

example, if you think of a container as a virtual machine, you might want to cram a lot of things in it, such as services, monitoring software and your application. Doing so could lead to a situation commonly called “the matrix of hell.” Don’t put many things into a single container image; instead, use many containers to achieve the full stack. In other words, you can keep your supporting service containers separate from your application container, and they can all be running on different operating systems and versions while being linked together.

Next, the way containers worked and behaved was largely undefined when Docker first popularized the technology. Many organizations wondered if containerization would really pay off, and some remain skeptical.

And while an engineering team might have extensive experience in implementing VM-based approaches, it might not have a conceptual understanding of how containers themselves work and behave. A key principle of container technology is that an image never changes, giving you an immutable starting point each time you run the image and the confidence that it will do the same thing each time you run it, no matter where you run it. To make changes, you create a new image and replace the current image with the newer version. This can be a challenging concept to embrace, until you see it in action.

These challenges have been largely overcome, however, as adoption spread and organizations began to realize the benefits of containers — or see their competitors realize them.

DevOps with Containers

Docker runs processes in isolated containers — processes that run on a local or remote host. When you execute the command `docker run`, the container process that runs is isolated: It has its own file system, its own

networking and its own isolated process tree separate from the host.

Essentially it works like this: A container image is a collection of file system layers and amounts to a fixed starting point. When you run an image, it creates a container. This container-based deployment capability is consistent from development machine to staging to QA to production — all the way through. When you have your application in a container, you can be sure that the code you're testing locally is exactly the same build artifact that goes into production. There are no changes in application runtime environments.

You once had specialized servers and were worried about them falling apart and having to replace them. Now servers are easily replaceable and can be scaled up or down — all your server needs to be able to do is run the container. It no longer matters which server is running your container, or whether that server is on premises, in the public cloud or a hybrid of both. You don't need an application server, web server or different specialized server for every application that's running. And if you lose a server, another server can run that same container. You can deploy any number of applications using the same tools and the same servers. Compartmentalization, consistency and standardized workflows have transformed deployments.

Containerization provided significant improvements to application deployment on each server. Instead of worrying about installing application dependencies on servers, they were included directly in the container image. This technology provided the foundation for transformative orchestration tooling such as Mesos and Kubernetes that would simplify deploying containers at scale.

Containers Evolved DevOps and the Profession

Developers were always connected to operations, whether they wanted to

be or not. If their application wasn't up and running, they were brought in to resolve problems. Google was one of the first organizations to introduce the concept of site reliability engineering, in which talented developers also have skill in the operations world. The book, [Site Reliability Engineering: How Google Runs Production Systems](#) (2016), describes best practices for building, deploying, monitoring and maintaining some of the largest software systems in the world, using a division of 50 percent development work and 50 percent operational work. This concept has taken off over the past two to three years as more organizations adopt DevOps practices in order to migrate to microservices and container-based architectures.

What began as two disparate job functions with crossover has now become its own job function. Operations teams are working with code bases; developers are working to deploy applications and are getting farther into the operational system. From an operational perspective, developers can look backward and read the CI file and understand the deployment processes. You can even look at Dockerfiles and see all the dependencies your application needs. It's simpler from an operational perspective to understand the code base.

So who exactly is this DevOps engineer? It's interesting to see how a specialization in DevOps has evolved. Some DevOps team members have an operational background, while others have a strong software development background. The thing that connects these diverse backgrounds is a desire for and an appreciation of system automation. Operations engineers gain development experience, and developers gain exposure to the behind-the-scenes ways the applications run. As this new specialization continues to evolve, next-generation DevOps tooling is continually being designed and built to accommodate changing roles and architectures in containerized infrastructure.

Running Containers with Kubernetes

In 2015, being able to programmatically “schedule” workloads into an application-agnostic infrastructure was the way forward. Today, the best practice is to migrate to some form of container orchestration.

Many organizations still use Docker to package up their applications, citing its consistency. Docker was a great step in the right direction, but it was a means to an end. In fact, the way containers were deployed wasn't transformative until orchestration tooling came about. Just as many container technologies existed before Docker, many container orchestration technologies preceded Kubernetes. One of the better known tools was Apache Mesos, a tool built by Twitter. Mesos does powerful things with regards to container orchestration, but it was — and still can be — difficult to set up and use. Mesos is still used by enterprises with scale and size, and it's an excellent tool for the right use case and scale.

Today, organizations are increasingly choosing to use Kubernetes instead of other orchestration tools. More and more, organizations are recognizing that containers offer a better solution than the more traditional tooling they had been using, and that Kubernetes is the best container deployment and management solution available. Let's examine these ideas further.

Introduction to Kubernetes

Kubernetes is a powerful, next generation, open source platform for automating the deployment, scaling and management of application containers across clusters of hosts. It can run any workload. Kubernetes provides exceptional developer user experience (UX), and the rate of innovation is phenomenal. From the start, Kubernetes' infrastructure promised to enable organizations to deploy applications rapidly at scale

and roll out new features easily while using only the resources needed. With Kubernetes, organizations can have their own Heroku running in their own public cloud or on-premises environment.

First released by Google in 2014, Kubernetes looked promising from the outset. Everyone wanted zero-downtime deployments, a fully automated deployment pipeline, auto scaling, monitoring, alerting and logging. Back then, however, setting up a Kubernetes cluster was hard. At the time, Kubernetes was essentially a do-it-yourself project with lots of manual steps. Many complicated decisions were — and are — involved: You have to generate certificates, spin up VMs with the correct roles and permissions, get packages onto those VMs and then build configuration files with cloud provider settings, IP addresses, DNS entries, etc. Add to that the fact that at first not everything worked as expected, and it's no surprise that many in the industry were hesitant to use Kubernetes.

Kubernetes 1.2, released in April 2016, included features geared more toward general-purpose usage. It was accurately touted as the next big thing. From the start, this groundbreaking open source project was an elegant, structured, real-world solution to containerization at scale that solves key challenges that other technologies didn't address. Kubernetes includes smart architectural decisions that facilitate the structuring of applications within containerization. Many things remain in your control. For example, you can decide how to set up, maintain and monitor different Kubernetes clusters, as well as how to integrate those clusters into the rest of your cloud-based infrastructure.

Kubernetes is backed by major industry players, including Amazon, Google, Microsoft and Red Hat. With over 14,000 individual contributors and ever increasing momentum, this project is here to stay.

In years past, think about how often development teams wanted visibility into operations deployments. Developers and operations teams have always been nervous about deployments because maintenance windows had a tendency to expand, causing downtime. Operations teams, in turn, have traditionally guarded their territory so no one would interfere with their ability to get their job done.

Then containerization and Kubernetes came along, and software engineers wanted to learn about it and use it. It's revolutionary. It's not a traditional operational paradigm. It's software driven, and it lends itself well to tooling and automation. Kubernetes enables engineers to focus on mission-driven coding, not on providing desktop support. At the same time, it takes engineers into the world of operations, giving development and operations teams a clear window into each other's worlds.

Kubernetes Is a Game Changer

Kubernetes is changing the game, not only in the way the work is done, but in who is being drawn to the field. Kubernetes has evolved into the standard for container orchestration. And the impacts on the industry have been massive.

In the past, servers were custom-built to run a specific application; if a server went down, you had to figure out how to rebuild it. Kubernetes simplifies the deployment process and improves resource utilization. As we stated previously, with Kubernetes your servers can be dumb — they don't care what they're running. Instead of running a specific application on a specific server, you can stack resources. A web server and a backend processing server might both run in Docker containers, for example. Let's say you have three servers, and five applications can run on each one. If one server goes down, you have redundancy because everything filters across.

Some benefits of Kubernetes:

- **Independently Deployable Services:** You can develop applications as a suite of independently deployable, modular services. Infrastructure code can be built with Kubernetes for almost any software stack, so organizations can create repeatable processes that are scalable across many different applications.
- **Deployment Frequency:** In the DevOps world, the entire team shares the same business goals and remains accountable for building and running applications that meet expectations. Deploying shorter units of work more frequently minimizes the amount of code you have to sift through to diagnose problems. The speed and simplicity of Kubernetes deployments enables teams to deploy frequent application updates.
- **Resiliency:** A core goal of DevOps teams is to achieve greater system availability through automation. With that in mind, Kubernetes is designed to recover from failure automatically. For example, if an application dies, Kubernetes will automatically restart it.
- **Usability:** Kubernetes has a well-documented API with simple, straightforward configuration that offers phenomenal developer UX. Together, DevOps practices and Kubernetes also allow businesses to deliver applications and features into users' hands quickly, which translates into more competitive products and more revenue opportunities.

Kubernetes Simplifies the Orchestration of Your Application

In addition to improving traditional DevOps processes, along with the speed, efficiency and resiliency commonly recognized as benefits of DevOps, Kubernetes solves new problems that arise with container and

microservices-based application architectures. Said another way, Kubernetes reinforces DevOps goals while also enabling new workflows that arise with microservices architectures.

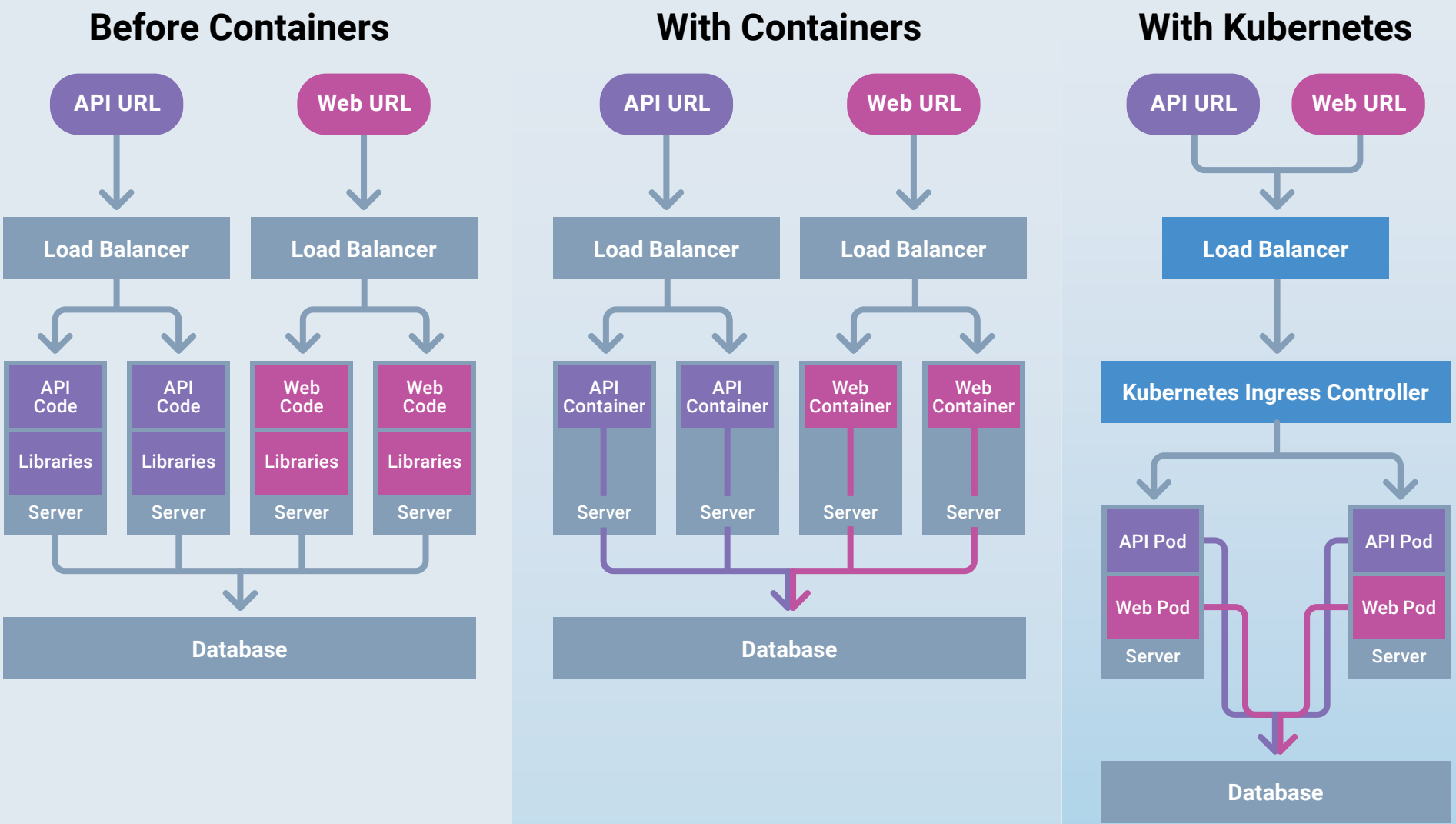
Powerful Building Blocks

Kubernetes uses pods as the fundamental unit of deployment. Pods represent a group of one or more containers that use the same storage and network. Although pods are often used to run only a single container, they have been used in some creative ways, including as a means to build a service mesh.

A common use of multiple containers in a single pod follows a sidecar pattern. With this pattern, a container would run beside your core application to provide some additional value. This is commonly used for

FIG 1.1: *With Kubernetes, pods are distributed across servers with load balancing and routing built in. Distributing application workloads in this way can dramatically increase resource utilization.*

The Evolution of Application Infrastructure



proxying requests, or even handling authentication.

With these powerful building blocks, it becomes quite straightforward to map services that may have been running in a virtual machine before containerization, into multiple containers running in the same pod.

Simplified Service Discovery

In one monolithic application, different services each have their own purpose, but self-containment facilitates communication. In a microservices architecture, microservices need to talk to each other — your user service needs to talk to your post service and address service and so on. Figuring out how these services can communicate simply and consistently is no easy feat.

With Kubernetes, a DevOps engineer defines a service — for example, a user service. Anything running in that same Kubernetes namespace can send a request to that service, and Kubernetes figures out how to route the request for you, making microservices easier to manage.

Centralized, Easily Readable Configuration

Kubernetes operates on a declarative model: You describe a desired state, and Kubernetes will try to achieve that state. Kubernetes has easily readable YAML files used to describe the state you want to achieve. With Kubernetes YAML configuration, you can define anything from an application load balancer to a group of pods to run your application. A deployment configuration might have three replicas of one of your applications' Docker containers and two different environment variables. This easy-to-read configuration is likely stored in a Git repository, so you can see any time that the configuration changes. Before Kubernetes, it was hard to know what was actually happening with interconnected systems across servers.

In addition to configuring the application containers running in your

cluster, or the endpoints that can be used to access them, Kubernetes can help with configuration management. Kubernetes has a concept called ConfigMap where you can define environment variables and configuration files for your application. Similarly, objects called secrets contain sensitive information and help define how your application will run. Secrets work much like ConfigMaps, but are more obscure and less visible to end users. [Chapter 2](#) explores all of this in detail.

Real-Time Source of Truth

Manual and scripted releases used to be extremely stressful. You had one chance to get it right. With the built-in deployment power of Kubernetes, anybody can deploy and check on delivery status using Kubernetes' unlimited deployment history: `kubectl rollout history`.

The Kubernetes API provides a real-time source of truth about deployment status. Any developer with access to the cluster can quickly find out what's happening with the delivery or see all commands issued. This permanent system audit log is kept in one place for security and historical purposes. You can easily learn about previous deployments, see the delta between deployments or roll back to any of the listed versions.

Simple Health Check Capability

This is a huge deal in your application's life cycle, especially during the deployment phase. In the past, applications often had no automatic restart if they crashed; instead, someone got paged in the middle of the night and had to restart them. Kubernetes, on the other hand, has automatic health checks, and if an application fails to respond for any reason, including running out of memory or just locking up, Kubernetes automatically restarts it.

To clarify, Kubernetes checks that your application is running, but it doesn't know how to check that it's running correctly. However,

Kubernetes makes it simple to set up health checks for your application. You can check the application's health in two ways:

1. **Using a liveness probe** that checks if an application goes from a healthy state to an unhealthy state. If it makes that transition, it will try to restart your application for you.
2. **Using a readiness probe** that checks if an application is ready to accept traffic. It won't get rid of previously working containers until the new containers are healthy. Basically, a readiness probe is a last line of defense that prevents a broken container from seeing the light of day.

Both probes are useful tools, and Kubernetes makes them easy to configure.

In addition, rollbacks are rare if you have a properly configured readiness probe. If all the health checks fail, a single one-line command will roll back that deployment for you and get you back to a stable state. It's not commonly used, but it's there if you need it.

Rolling Updates and Native Rollback

To build further off the idea of a real-time source of truth and health check capabilities, another key feature of Kubernetes is rolling updates with the aforementioned native rollback. Deployments can and should be frequent without fear of hitting a point of no return. Before Kubernetes, if you wanted to deploy something, a common deployment pattern involved the server pulling in the newest application code and restarting your application. The process was risky because some features weren't backwards compatible — if something went wrong during the deployment, the software became unavailable. For example, if the server found new code, it would pull in those updates and try to restart the application with the new code. If something failed in that pipeline, the application was likely dead. The rollback procedure was anything but straightforward.

These workflows were problematic until Kubernetes. Kubernetes solves this problem with a deployment rollback capability that eliminates large maintenance windows and anxiety about downtime. Since Kubernetes 1.2, the deployment object is a declarative manifest containing everything that's being delivered, including the number of replicas being deployed and the version of the software image. These items are abstracted and contained within a deployment declaration. Such manifest-based deployments have spurred new CD workflows and are an evolving best practice with Kubernetes.

Before Kubernetes shuts down existing application containers, it will start spinning up new ones. Only when the new ones are up and running correctly does it get rid of the old, stable release. Let's say Kubernetes doesn't catch a failed deployment — the app is running, but it's in some sort of error state that Kubernetes doesn't detect. In this case, DevOps engineers can use a simple Kubernetes command to undo that deployment. Furthermore, you can configure it to store as few as two changes or as many revisions as you want, and you can go back to the last deployment or many deployments earlier, all with an automated, simple Kubernetes command. This entire concept was a game-changer. Other orchestration frameworks don't come close to handling this process in as seamless and logical a way as Kubernetes.

Simplified Monitoring

While on the surface it might seem that monitoring Kubernetes would be quite complex, there has been a lot of development in this space. Although Kubernetes and containers add some levels of complexity to your infrastructure, they also ensure that all your applications are running in consistent pods and deployments. This consistency enables monitoring tools to be simpler in many ways.

Prometheus is an example of an open source monitoring tool that has

become very popular in the cloud-native ecosystem. This tool provides advanced monitoring and alerting capabilities, with excellent Kubernetes integrations.

When monitoring Kubernetes, there are a few key components to watch: Kubernetes nodes (servers); Kubernetes system deployments, such as DNS or networking; and, of course, your application itself. There are many monitoring tools that will simplify monitoring each of these components.

Kubernetes and CI/CD Complement Each Other

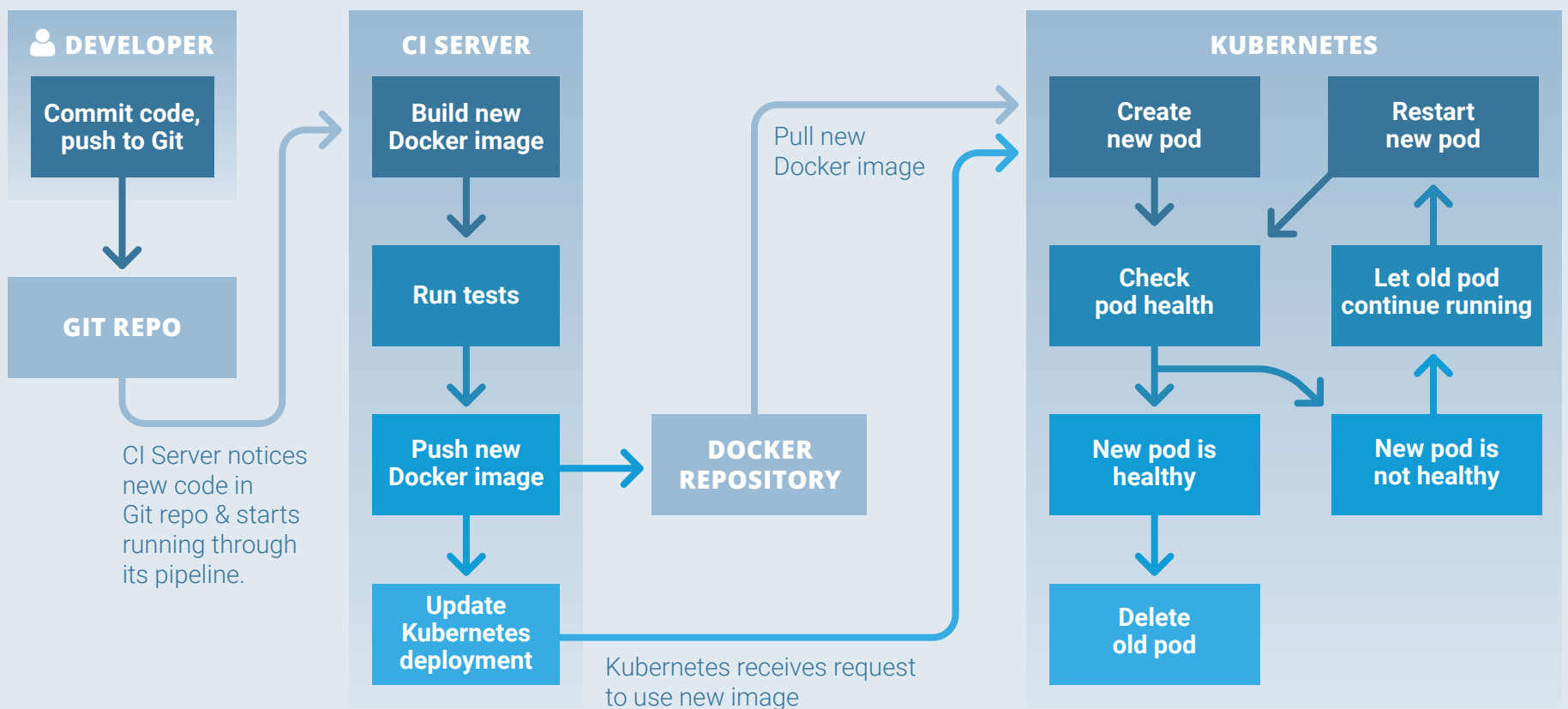
Setting up a CI/CD pipeline on top of Kubernetes will speed up your release life cycle — enabling you to release multiple times a day — and enable nimble teams to iterate quickly. With Kubernetes, builds become a lot faster. Instead of spinning up entirely new servers, your build process is quick, lightweight and straightforward.

Development speeds up when you don't have to worry about building and deploying a monolith in order to update everything. By splitting a monolith into microservices, you can instead update pieces — this service or that. Part of a good CI/CD workflow should also include a strong test suite. While not unique to Kubernetes, a containerized approach can make tests more straightforward to run. If your application tests depend on other services, you can run your tests against those containers, simplifying the testing process. A one-line command is usually all you need to update a Kubernetes deployment.

In a CI/CD workflow, ideally you run many tests. If those tests fail, your image will never be built, and you'll never deploy that container.

However, if testing fails to uncover issues, Kubernetes offers better protection because Kubernetes simplifies zero-downtime deployment. For a long time, deployments meant downtime. Operations teams used to

CI/CD Pipeline Workflow with Kubernetes



Source: ReactiveOps

© 2018 THE NEW STACK

FIG 1.2: Before Kubernetes shuts down existing pods, it will start spinning up new ones. Only when the new ones are up and running correctly does it get rid of the old, stable release. Such rolling updates and native rollback features are a game-changer for DevOps.

handle deployment efforts manually or via scripting, a live process that could take hours, if not all night. Accordingly, one of the fears of CI/CD is that a deployment will break and a site will go down.

Kubernetes' zero-downtime deployment capability relieves anxieties about maintenance windows, making schedule delays and downtime a thing of the past — and saving money in the process. It also keeps everyone in the loop while meeting the needs of development, operations and business teams. The revolutionary Kubernetes deployment object has built-in features that automate this operations effort.

In particular, the aforementioned tests and health checks can prevent bad code from reaching production. As part of a rolling update, Kubernetes spins up separate new pods running your application while the old ones are still running. When the new pods are healthy, Kubernetes gets rid of

the old ones. It's a smart, simple concept, and it's one less thing you have to worry about for each application and in your CI/CD workflow.

Complementary Tools

As part of the incredible momentum Kubernetes has seen, a number of DevOps tools have emerged that are particularly helpful in developing CI/CD workflows with Kubernetes. Bear in mind that CI/CD tools and practices are still evolving with the advent of cloud-native deployments on Kubernetes. No single tool yet offers the perfect solution for managing cloud-native applications from build through deployment and continuous delivery. Although there are far too many to mention here, it's worth highlighting a few DevOps tools that were purpose-built for cloud-native applications:

- **Draft:** This tool from Microsoft targets developer workflows. With a few simple commands, Draft can containerize and deploy an application to Kubernetes. The automated containerization of applications here can be quite powerful. Draft uses best practices for popular frameworks and languages to build images and Kubernetes configuration that will work in most cases.
- **Helm:** Known as the Kubernetes package manager, this framework simplifies deploying applications to Kubernetes. Deployment configuration for many popular projects is available in well maintained "Charts." This means that `helm install prometheus` is all that's needed to get a project like Prometheus running in your cluster. Helm can also provide the same kind of conveniences when deploying your own custom applications.
- **Skaffold:** Similar to Draft, this is a new tool from Google that enables exciting new development workflows. In addition to supporting more standard CI/CD workflows, this has an option to build and deploy code

to a Kubernetes development environment each time the code changes locally. This tool is highly configurable, and even supports using Helm for deployments.

- **Spinnaker:** This open source continuous delivery platform was developed by Netflix to handle CD operations at high scale over its cloud network. It is a cloud-native pipeline management tool that supports integrations into all the major cloud providers: Amazon Web Services (AWS), Azure, Google Cloud Platform and OpenStack. It natively supports Kubernetes deployments, but its scope extends much farther beyond Kubernetes.

Extensions of DevOps

Continuous deployment of cloud-native applications has transformed the way teams collaborate. Transparency and observability at every stage of development and deployment are increasingly the norm. It's probably no surprise then that GitOps and SecOps, both enabled by cloud-native architecture, are building on current DevOps practices by providing a single source of truth for changes to the infrastructure and changes to security policies and rules. The sections below highlight these evolutionary developments.

GitOps

Git is becoming the standard for distributed version control, wherein a Git repository contains the entire system: code, config, monitoring rules, dashboards and a full audit trail. [GitOps](#) is an iteration of DevOps, wherein Git is a single source of truth for the whole system, enabling rapid application development on cloud-native systems, and using Kubernetes in particular. "GitOps" is a term developed by Weaveworks to describe DevOps best practices in the age of Kubernetes, and it strongly emphasizes a declarative infrastructure.

The fundamental theorem of GitOps is that if you can describe it, you can automate it. And if you can automate it, you can control and accelerate it. The goal is to describe everything — policies, code, configuration and monitoring — and then version control everything.

With GitOps, your code should represent the state of your infrastructure. GitOps borrows DevOps logic:

- All code must be version-controlled.
- Configuration is code.
- Configuration must also be version-controlled.

The idea behind GitOps is transparency. A declarative environment captures state, enabling you to compare the observed state and the desired state easily. In fact, you can observe the system at all times. In short, every service has two sources of truth: the desired state of the system and the observed state of the system.

In the truest sense of GitOps, Git maintains a repository that describes your infrastructure and all of your Kubernetes configuration, and your local copy of code gives you a complete version control repository. When you push new code or configuration to Git, something on the other side listens for this new push and makes the changes for you. All changes happen the same way. All of your infrastructure and configuration live in a centralized repository, and every single change is driven by a commit and push of that repository.

How it works: Code is committed and pushed to GitHub, and then you have a CI/CD workflow listening on the other side, and that CI/CD workflow makes those changes and commits those changes in the configuration. The key difference: Instead of engineers interacting with Kubernetes or the system configuration directly — say, using Kubernetes CLI — they're doing

everything through Git: writing configuration, pushing configuration and then applying those changes through the CI/CD workflow.

The three key goals of GitOps include:

1. **Pipelines:** Build completely automated CI/CD pipelines, enabling Git to become a source of truth for the desired system state.
2. **Observability:** Implement 24/7 monitoring, logging and security — observe and measure every service pull request to gain a holistic view of the current system state.
3. **Control:** Version control everything and create a repository that contains a single source of truth for recovery purposes.

With GitOps, Weaveworks is seeing people go from a few deployments a week using CI systems to 30 to 50 deployments per day. In addition, DevOps teams are fixing bugs twice as fast. Managing the state of the code in Git versus Kubernetes allows for better tracking and recovery. It also allows for continuous experimentation, such as A/B testing and response to customer ideas, on the Kubernetes architecture.

SecOps

Security teams traditionally hand off security test results and vulnerability scans to operations teams for review and implementation, often as an application is being deployed. So long as an application is running and performing as expected, security's involvement in the process gets a green light. However, information exchange and approval cycles can lead to delays and slow what would otherwise be an agile DevOps workflow. It's not surprising that this occurs as the two teams are dealing with two very different sets of goals. Operations tries to get the system running in as straightforward and resilient a manner as possible. Security, on the other hand, seeks to control the environment — the fewer things running, the better.

In reality, these late-stage handoffs between operations and security are problematic for many reasons, not the least of which is that the insights and expertise of both teams are siloed rather than shared. As a result, potential threats can grow into showstoppers, and security-related problems often fester, going undetected for longer periods than warranted.

When organizations don't have a mechanism for communicating and transferring key security data on an ongoing basis, those organizations inevitably struggle to mitigate security risks, prioritize and remediate security threats and vulnerabilities, and ultimately protect their application environment. However, it's not necessary for organizations to sacrifice security in order to maintain uptime and performance. That's where SecOps comes in.

SecOps bridges the efforts of security and operations teams in the same way that DevOps bridges the efforts of software developers and operations teams. Just as a DevOps approach allows product developers to strategically deploy, manage, monitor and secure their own applications, a SecOps approach gives engineers a window into both operations and security issues. It's a transition from individual tacticians — system administrators and database administrators — to more strategic roles within the organization. These teams share priorities, processes, tools and, most importantly, accountability, giving organizations a centralized view of vulnerabilities and remediation actions while also automating and accelerating corrective actions.

In a GitOps approach, all your configuration and infrastructure are stored in a central Git repository. DevOps engineers write development and deployment policies, whereas security engineers write security firewall rules and network policies. And all of these rules end up in the same repository. Collaboration among these teams — DevOps and SecOps, or

SecDevOps — ups the ante, increasing efficiency and transparency into what has happened and what should happen in the future.

In replacing disconnected, reactive security efforts with a unified, proactive CI/CD-based security solution for both cloud and on-premises systems, SecOps gains a more cohesive team from a diversity of backgrounds working toward a common goal: frequent, fast, zero-downtime, secure deployments. This goal empowers both operations and security to analyze security events and data with an eye toward reducing response times, optimizing security controls and checking and correcting vulnerabilities at every stage of development and deployment. Blurring the lines between the operations and security teams brings greater visibility into any development or deployment changes warranted, along with the potential impacts of those changes.

Conclusion

The story of DevOps and Kubernetes is one of continued, fast-paced evolution. For example, in the current state of DevOps, technologies that may be only a few years old can start to feel ancient. The industry is changing on a dime.

Kubernetes is still very new and exciting, and there's incredible demand in the market for it. Organizations are leveraging DevOps to migrate to the cloud, automate infrastructure and take Software as a Service (SaaS) and web applications to the next level. Consider what you might accomplish with higher availability, autoscaling and a richer feature set. Kubernetes has vastly improved CI/CD workflows, allowing developers to do amazing things they couldn't do before. GitOps, in turn, offers a centralized configuration capability that makes Kubernetes easy. With a transparent and centralized configuration, changes aren't being individually applied willy nilly, but are instead going through the same pipeline. Today, product

teams are centralizing development, operations and security, working towards the same business goals: easier and faster deployments, less downtime, fewer outages and faster time to recovery.

Organizations are increasingly turning to Kubernetes because they have determined it's the right tool for the job. Kubernetes just works — right out of the box. Kubernetes makes it hard to do the wrong thing and easy to do the right thing. More than half of Fortune 100 companies are using Kubernetes, [according to RedMonk](#). But the story doesn't end there. Mid-sized companies use Kubernetes. Ten-person startups use Kubernetes. Kubernetes isn't just for enterprises. It's the real deal for forward-thinking companies of all sizes. And it's completely transforming the way software is innovated, designed, built and deployed.

DevOps and Kubernetes are the future. Together, they just make good business sense.

THE BEST CI/CD TOOL FOR KUBERNETES DOESN'T EXIST



There is no single, best set of tools for continuous integration / continuous delivery (CI/CD) with Kubernetes — each organization will use the tools that are best suited for its specific use case.

“The hardest thing [about running on Kubernetes] is gluing all the pieces together. You need to think more holistically about your systems and get a deep understanding of what you’re working with,” says [Chris Short](#), a DevOps consultant and Cloud Native Computing Foundation (CNCF) ambassador.

We talk with Short and [Ihor Dvoretzkyi](#), developer advocate at the CNCF, about the trends they’re seeing in DevOps and CI/CD with Kubernetes, the role of the Kubernetes community in improving CI/CD and some of the challenges organizations face as they consider the plethora of tools available today. [Listen on SoundCloud »](#)



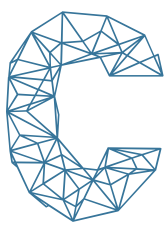
Chris Short has spent more than two decades in various IT disciplines, and has been an active proponent of open source solutions throughout his time in the private and public sectors. Read more at [chrisshort.net](#), or follow his DevOps, cloud native, and open source-focused newsletter [DevOps'ish](#).



Ihor Dvoretzkyi is a developer advocate at the Cloud Native Computing Foundation. He is a product manager for Kubernetes, co-leading the Product Management Special Interest Group, focused on enhancing Kubernetes as an open source product. In addition, he participates in the Kubernetes release process as a features lead.

CLOUD-NATIVE APPLICATION PATTERNS

by **JANAKIRAM MSV**



Container technologies tell a story about a new wave of developers. Their ranks are filled with people who are creating microservices that run on sophisticated underlying infrastructure technologies. Developers are drawn to the utility and packaging capabilities in containers which improve efficiency and fit with modern application development practices.

Developers use application architectures with container technologies to develop services that reflect an organizational objective. Kubernetes runs under the application architecture and is used to run services across multiple clusters, providing the abstraction of orchestration to manage microservices following DevOps practices. The flexibility of today's cloud-native, modern services running on infrastructure managed by software gives developers capabilities that had previously not been possible. Developers now have resources for connecting more endpoints through integrations that feed into declarative infrastructure. Dynamic, declarative infrastructure is now a foundation for development, and will increasingly serve as a resource for event-driven automation in modern application architectures.

In all of this, the container can be defined as a unit. Each unit holds code, a payload that in more complex operations gets orchestrated across distributed architectures and infrastructure managed by cloud services. More developers are now testing how to optimize the resources available on different types of infrastructure to take advantage of the benefits offered by containers and virtualization.

Cloud native is a term used to describe container-based environments. Cloud-native technologies are used to develop applications built with services packaged in containers, deployed as microservices, and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows. It's important to note that in this equation, the teams and processes that allow for increased speed and agility are as important as the technology itself.

“Cloud Native is structuring teams, culture and technology to utilize automation and architectures to manage complexity and unlock velocity.”

- Joe Beda, CTO and co-founder of Heptio.

Where operations teams would manage the infrastructure resource allocations to traditional applications manually, cloud-native applications are deployed on infrastructure that abstracts the underlying compute, storage and networking primitives. Developers and operators dealing with this new breed of applications don't directly interact with application programming interfaces (APIs) exposed by infrastructure providers. Instead, the orchestrator handles resource allocation automatically,

according to policies set out by DevOps teams. The controller and scheduler, which are essential components of the orchestration engine, handle resource allocation and the life cycle of applications. Cloud-native platforms, like Kubernetes, expose a flat network that is overlaid on existing networking topologies and primitives of cloud providers. Similarly, the native storage layer is often abstracted to expose logical volumes that are integrated with containers. Operators can allocate storage quotas and network policies that are accessed by developers and resource administrators. The infrastructure abstraction not only addresses the need for portability across cloud environments, but also lets developers take advantage of emerging patterns to build and deploy applications. Orchestration managers become the deployment target, irrespective of the the underlying infrastructure that may be based on physical servers or virtual machines, private clouds or public clouds.

Kubernetes is an ideal platform for running contemporary workloads designed as cloud-native applications. It's become the de facto operating system for the cloud, in much the same way Linux is the operating system for the underlying machines. Our author for this chapter is Janakiram MSV, the Principal Analyst at Janakiram & Associates and an adjunct faculty member at the International Institute of Information Technology. He is an Ambassador for the Cloud Native Computing Foundation — home of the Kubernetes open source project — and was also one of the first Certified Kubernetes Administrators and Certified Kubernetes App Developers. Here, he brings his considerable expertise consulting with organizations on their cloud strategies and Kubernetes implementations to map the building blocks of cloud-native applications with the constructs and primitives of Kubernetes. The chapter acts as a guide to choosing the right Kubernetes object for each of the components of a mature cloud-native application.

As long as developers follow best practices of designing and developing

software as a set of microservices that comprise cloud-native applications, DevOps teams will be able to package and deploy them in Kubernetes. This chapter is intended to help guide DevOps teams deploying cloud-native applications in Kubernetes.

10 Key Attributes of Cloud-Native Applications

Before taking a look at the big picture of cloud-native application design, let's review the ten key attributes of cloud-native applications.

- 1. Packaged as lightweight containers:** Cloud-native applications are a collection of independent and autonomous services that are packaged as lightweight containers. Unlike virtual machines, containers can scale out and scale in rapidly. Since the unit of scaling shifts to containers, infrastructure utilization is optimized.
- 2. Developed with best-of-breed languages and frameworks:** Each service of a cloud-native application is developed using the language and framework best suited for the functionality. Cloud-native applications are polyglot; services use a variety of languages, runtimes and frameworks. For example, developers may build a real-time streaming service based on WebSockets, developed in Node.js, while choosing Python and Flask for exposing the API. The fine-grained approach to developing microservices lets them choose the best language and framework for a specific job.
- 3. Designed as loosely coupled microservices:** Services that belong to the same application discover each other through the application runtime. They exist independent of other services. Elastic infrastructure and application architectures, when integrated correctly, can be scaled out with efficiency and high performance.

Loosely coupled services allow developers to treat each service independent of the other. With this decoupling, a developer can focus on the core functionality of each service to deliver fine-grained functionality. This approach leads to efficient life cycle management of the overall application, because each service is maintained independently and with clear ownership.

4. Centered around APIs for interaction and collaboration: Cloud-native services use lightweight APIs that are based on protocols such as representational state transfer (REST), Google's open source remote procedure call (gRPC) or NATS. REST is used as the lowest common denominator to expose APIs over hypertext transfer protocol (HTTP). For performance, gRPC is typically used for internal communication among services. NATS has publish-subscribe features which enable asynchronous communication within the application.

5. Architected with a clean separation of stateless and stateful services: Services that are persistent and durable follow a different pattern that assures higher availability and resiliency. Stateless services exist independent of stateful services. There is a connection here to how storage plays into container usage. Persistence is a factor that has to be increasingly viewed in context with state, statelessness and — some would argue — micro-storage environments.

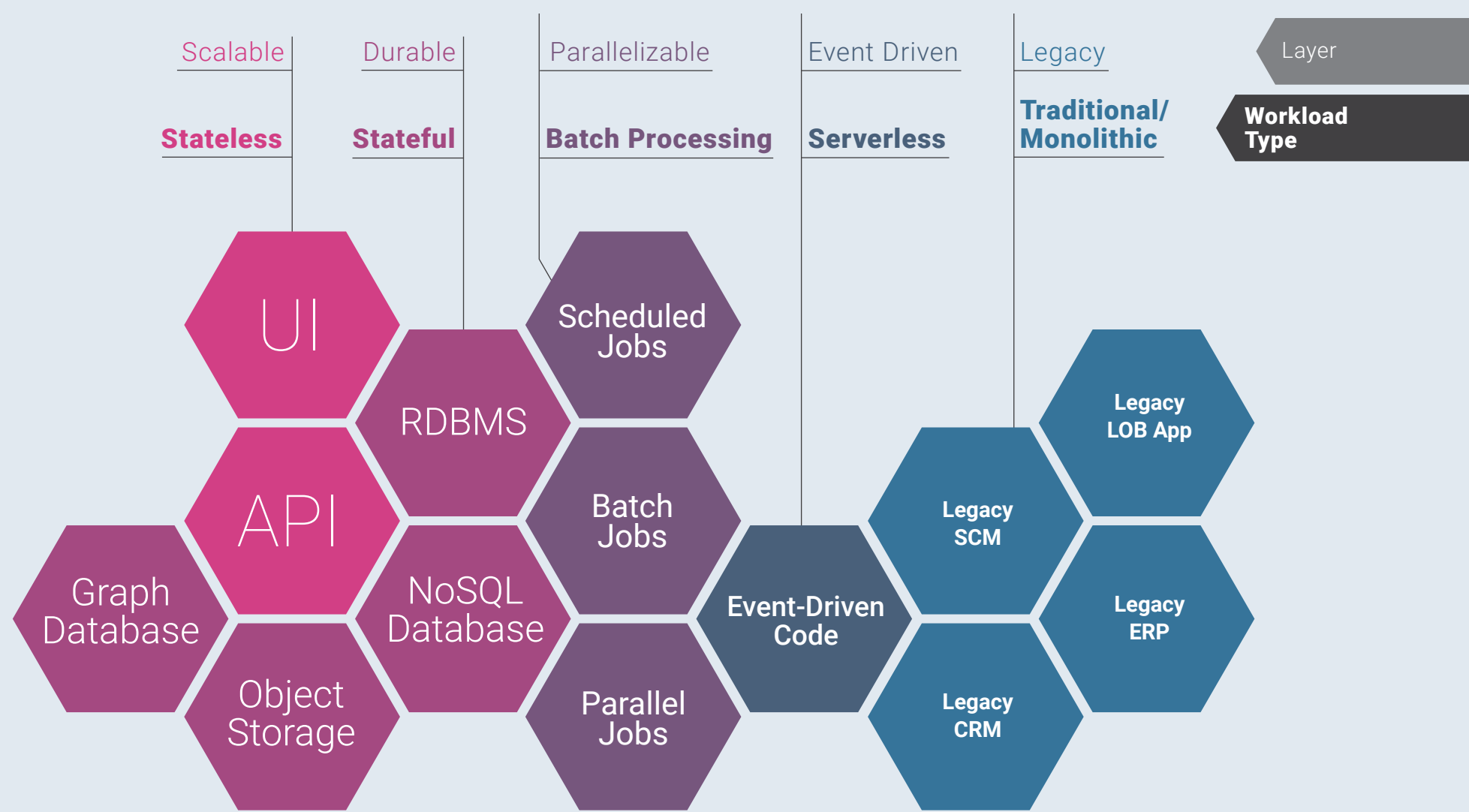
6. Isolated from server and operating system dependencies: Cloud-native applications don't have an affinity for any particular operating system or individual machine. They operate at a higher abstraction level. The only exception is when a microservice needs certain capabilities, including solid-state drives (SSDs) and graphics processing units (GPUs), that may be exclusively offered by a subset of machines.

- 7. Deployed on self-service, elastic, cloud infrastructure:** Cloud-native applications are deployed on virtual, shared and elastic infrastructure. They may align with the underlying infrastructure to dynamically grow and shrink — adjusting themselves to the varying load.
- 8. Managed through agile DevOps processes:** Each service of a cloud-native application goes through an independent life cycle, which is managed through an agile DevOps process. Multiple continuous integration/continuous delivery (CI/CD) pipelines may work in tandem to deploy and manage a cloud-native application.
- 9. Automated capabilities:** Cloud-native applications can be highly automated. They play well with the concept of infrastructure as code. Indeed, a certain level of automation is required simply to manage these large and complex applications.
- 10. Defined, policy-driven resource allocation:** Finally, cloud-native applications align with the governance model defined through a set of policies. They adhere to policies such as central processing unit (CPU) and storage quotas, and network policies that allocate resources to services. For example, in an enterprise scenario, central IT can define policies to allocate resources for each department. Developers and DevOps teams in each department have complete access and ownership to their share of resources.

Overview of Cloud-Native Application Design

Cloud-native applications are composed of various logical layers, grouped according to functionality and deployment patterns. Each layer runs specific microservices designed to perform a fine-grained task. Some of

Mapping Layers to Cloud-Native Workloads



Source: Janakiram MSV

© 2018 THE NEW STACK

FIG 2.1: Today’s modern application architectures bridge with monolithic legacy systems.

these microservices are stateless, while others are stateful and durable. Certain parts of the application may run as batch processes. Code snippets may be deployed as functions that respond to events and alerts.

The depiction here attempts to identify layers of a cloud-native application. Though they are grouped together for representation, each layer is independent. Unlike traditional three-tier applications which are stacked in a hierarchy, cloud-native applications operate in a flat structure with each service exposing an API.

The **scalable layer** runs stateless services that expose the API and user experience. This layer can dynamically expand and shrink depending on the usage at runtime. During the scale-out operation, where more instances of the services are run, the underlying infrastructure may also scale out to match the CPU and memory requirements. An autoscale

policy is implemented to evaluate the need to perform scale-in and scale-out operations.

The **durable layer** has stateful services that are backed by polyglot persistence. It is polyglot because of the variety of databases that may be used for persistence. Stateful services rely on traditional relational databases, NoSQL databases, graph databases and object storage. Each service chooses an ideal datastore aligned with the structure of stored data. These stateful services expose high-level APIs that are consumed by both — the services from the scalable and durable layers.

Apart from stateless and stateful layers, there are scheduled jobs, batch jobs and parallel jobs that are classified as the **parallelizable layer**. For example, scheduled jobs may run extract, transform, load (ETL) tasks once per day to extract the metadata from the data stored in object storage and to populate a collection in the NoSQL database. For services that need scientific computing to perform machine learning training, the calculations are run in parallel. These jobs interface with the GPUs exposed by the underlying infrastructure.

To trigger actions resulting from events and alerts raised by any service in the platform, cloud-native applications may use a set of code snippets deployed in the **event-driven layer**. Unlike other services, the code running in this layer is not packaged as a container. Instead, functions written in languages such as Node.js and Python are deployed directly. This layer hosts stateless functions that are event driven.

Cloud-native applications also interoperate with existing applications at the **legacy layer**. Legacy, monolith applications — such as enterprise resource planning, customer relationship management, supply chain management, human resources and internal line-of-business applications — are accessed by services.

Enterprises will embrace microservices for building API layers and user interface (UI) frontends that will interoperate with existing applications. In this scenario, microservices augment and extend the functionality of existing applications. For example, they may have to talk to the relational database that is powering a line-of-business application, while delivering an elastic frontend deployed as a microservice.

Each service of a cloud-native application exposes a well-defined API, which is consumed by other services. For intra-service communication, protocols like gRPC or NATS are preferred due to their efficient compression and binary compatibility. The REST protocol is used for exposing services that interact with the external world.

DevOps teams map the deployment and communication patterns with the primitives exposed by cloud-native platforms such as Kubernetes. They are expected to package, deploy and manage these services running in a production environment. The next section helps with the alignment and mapping of the workload patterns with Kubernetes primitives.

Mapping Cloud-Native Workloads to Kubernetes Objects

Kubernetes is more than just a container manager. It's a platform designed to handle a variety of workloads packaged in any number of containers and combinations. There are multiple controllers built into Kubernetes that map to the layers of cloud-native architecture.

DevOps engineers can think of Kubernetes controllers as the means for dictating the infrastructure needs of the various workloads your team is running. They can define the desired configuration state through a declarative approach. For example, a container/pod deployed as a part of a ReplicationController is guaranteed to be available all the time. A

container packaged as a DaemonSet is guaranteed to run on every node of the cluster. The declarative approach enables DevOps teams to take advantage of paradigms such as infrastructure as code. Some of the deployment patterns discussed below follow the principles of immutable infrastructure, where each new rollout results in an atomic deployment.

The control plane of Kubernetes constantly tracks the deployments to ensure that they are adhering to the desired configuration state defined by DevOps.

The fundamental unit of deployment in Kubernetes is a pod. It is the basic building block of Kubernetes, which is the smallest and simplest unit in the Kubernetes object model. A pod represents a running process on the cluster. Irrespective of a service being stateful or stateless, it is always packaged and deployed as a pod.

A controller can create and manage multiple pods within the cluster, handling replication that provides self-healing capabilities at cluster scope. For example, if a node fails, the controller might automatically replace the pod by scheduling an identical replacement on a different node.

Kubernetes comes with multiple controllers to handle the desired state of pods. ReplicationController, Deployment, DaemonSet and StatefulSet are a few examples of controllers. Kubernetes controllers use a pod template that is provided to create the pods for which it is responsible to maintain the desired state. Pods, like other Kubernetes objects, are defined in a YAML file and submitted to the control plane.

When running cloud-native applications in Kubernetes, operators need to understand the use cases addressed by controllers to get the most out of the platform. This helps them in defining and maintaining the desired state of configuration of the application.

Each of the patterns explained in the previous section maps to specific Kubernetes controllers which allow more precise, fine-grained control of workloads on Kubernetes, but in an automated fashion.

The declarative configuration of Kubernetes encourages an immutable infrastructure. The deployments are tracked and managed by the control plane to ensure that the desired configuration state is maintained throughout the application life cycle. When compared to traditional deployments based on virtual machines, DevOps engineers will spend significantly less time maintaining workloads. An effective CI/CD strategy that takes advantage of Kubernetes primitives and deployment patterns frees operators from performing mundane tasks.

Scalable Layer: Stateless Workloads

Stateless workloads are packaged and deployed as a ReplicaSet in Kubernetes. A ReplicationController forms the basis of a ReplicaSet, which ensures that a specified number of pod replicas are always running at any given time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

If there are too many pods, the ReplicationController may terminate the extra pods. If there are too few, the ReplicationController proceeds to launch additional pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted or terminated. The pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, it is recommended to use a ReplicationController even if the application requires only a single pod.

A simple use case is to create one ReplicationController object to reliably run one instance of a pod indefinitely. A more complex use case is to run several identical replicas of a scale-out service, such as web servers.

DevOps teams and operators package stateless workloads as ReplicationControllers when deploying in Kubernetes.

In the recent versions of Kubernetes, ReplicaSets replaced ReplicationControllers. Both of them address the same scenario, but ReplicaSets use a [set-based label selector](#) which makes it possible to use complex queries based on annotations. Additionally, Deployments in Kubernetes rely on ReplicaSets.

Deployments are an abstraction of ReplicaSets. When a desired state is declared in the Deployment object, the Deployment controller changes the actual state to the desired state at a controlled rate.

Deployments are highly recommended to manage stateless services of cloud-native applications. Though services can be deployed as pods and ReplicaSets, Deployments make upgrading and patching your application easier. DevOps teams can upgrade a pod in place using a Deployment, which cannot be done with a ReplicaSet. This makes it possible to roll out a new version of an application with minimal downtime. Deployments bring Platform as a Service (PaaS)-like capabilities to application management.

Durable Layer: Stateful Workloads

Stateful workloads can be classified into two categories: services that need persistent storage (single instance) and services that need to run in a highly reliable and available mode (replicated multi-instance). A pod that needs access to a durable storage backend is very different from a set of pods that run a cluster for a relational database. While the former needs long-term, durable persistence, the latter needs high availability of the workload. Kubernetes addresses both scenarios.

Individual pods can be backed by volumes that expose underlying storage to the services. The volume may be mapped to an arbitrary node on

which the pod is scheduled. If multiple pods are scheduled across different nodes of the cluster and need to share the backend, a distributed file system such as Network File System (NFS) or Gluster is configured manually before deploying applications. Modern storage drivers available within the cloud-native ecosystem offer container-native storage where the file system itself is exposed through containers. Use this configuration when pods just need persistence and durability.

For scenarios where high availability is expected, Kubernetes offers StatefulSets — a specialized set of pods that guarantees the ordering and uniqueness of pods. This is especially useful in running primary/secondary — previously known as master/slave — configurations of database clusters.

Like a Deployment, a StatefulSet manages pods that are based on an identical container specification. Unlike a Deployment, a StatefulSet maintains a unique identity for each of its pods. These pods are created from the same spec, but are not interchangeable: Each pod has a persistent identifier that it maintains across any rescheduling.

StatefulSets are useful for workloads that require one or more of the following:

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, graceful deletion and termination.
- Ordered, automated rolling updates.

Kubernetes treats StatefulSets differently than other controllers. When pods of a StatefulSet are being scheduled with N replicas, they are created sequentially, in order from 0 to N-1. When pods of a StatefulSet are being

deleted, they are terminated in reverse order, from N-1 to 0. Before a scaling operation is applied to a pod, all of its predecessors must be running and ready. Kubernetes ensures that before a pod is terminated, all of its successors are completely shut down.

StatefulSets are recommended when services need to run clusters of Cassandra, MongoDB, MySQL, PostgreSQL or any database workloads with a high availability requirement.

Not every persistent workload needs to be a StatefulSet. Certain containers rely on a durable storage backend to store data. For adding persistence to these type of applications, pods may rely on volumes backed by either host-based storage or container-native storage backends.

Parallelizable Layer: Batch Processing

Kubernetes has built-in primitives for batch processing, which is useful for executing run to completion jobs or scheduled jobs.

Run to completion jobs are typically used for running processes that need to perform an operation and exit. A big data workload that runs until the data is processed is an example of such a job. Another example is a job that processes each message in a queue until the queue becomes empty.

A Job is a controller that creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the Job itself is complete. Deleting a Job will clean up the pods it created.

A Job can also be used to run multiple pods in parallel, which makes it ideal for machine learning training jobs. Jobs also support parallel processing of a set of independent but related work items.

When Kubernetes runs on hardware with GPUs, machine learning training can take advantage of Jobs. Emerging projects such as Kubeflow — a project dedicated to making deployment of machine learning on Kubernetes simple, portable and scalable — will expose primitives to package machine learning training as Jobs.

Apart from running parallelized jobs, there may be a need to run scheduled jobs. Kubernetes exposes CronJobs that can run once at a specified point in time or periodically at a specified point in time. A CronJob object in Kubernetes is similar to one line of a crontab (cron table) file in Unix. It runs a job periodically on a given schedule, written in cron format.

CronJobs are especially useful for scheduling periodic jobs such as database backups or sending emails.

Event-Driven Layer: Serverless

Serverless computing refers to the concept of building and running applications that do not require server management. It describes a more fine-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled and billed in response to the exact demand needed at the moment.

Functions as a Service (FaaS) runs within the context of serverless computing to provide event-driven computing. Developers run and manage application code with functions that are triggered by events or HTTP requests. Developers deploy small units of code to the FaaS, which are executed as needed as discrete actions, scaling without the need to manage servers or any other underlying infrastructure.

Though Kubernetes doesn't have an integrated event-driven primitive that responds to alerts and events raised by other services, there are efforts to bring event-driven capabilities. The [Cloud Native Computing Foundation](#),

the custodian of Kubernetes, has a serverless working group focused on these efforts. Open source projects such as [Apache OpenWhisk](#), [Fission](#), [Kubeless](#), [OpenFaaS](#) and [Oracle's Fn](#) can be run within a Kubernetes cluster as the event-driven, serverless layer.

Code deployed in the serverless environment is fundamentally different from the code packaged as pods. It consists of autonomous functions that can be wired to one or more events that may trigger the code.

When event-driven computing — serverless computing — becomes an integral part of Kubernetes, developers will be able to deploy functions that respond to both internal events generated by the Kubernetes control plane along with custom events raised by applications service.

Legacy Layer: Headless Services

Even after your organization is regularly building and deploying applications using a microservices architecture into containers on the cloud, there may be applications that continue to live outside of Kubernetes. Cloud-native applications and services will have to interact with those traditional, monolithic applications.

The legacy layer exists for interoperability, to expose a set of headless services pointing to the monolithic applications. Headless services allow developers to reduce coupling to the Kubernetes system by allowing them freedom to do discovery their own way. Headless services in Kubernetes are different from ClusterIP, NodePort and LoadBalancer types of services. They don't have an internet protocol (IP) address assigned to them, but have a domain name system (DNS) entry that points to an external endpoint such as API servers, web servers and databases. The legacy layer is a logical interoperability layer that maintains DNS records to well-known, external endpoints.

Each layer of a microservices application can be mapped to one of the

controllers of Kubernetes. Depending on the pattern they wish to deploy, DevOps teams can choose the appropriate option.

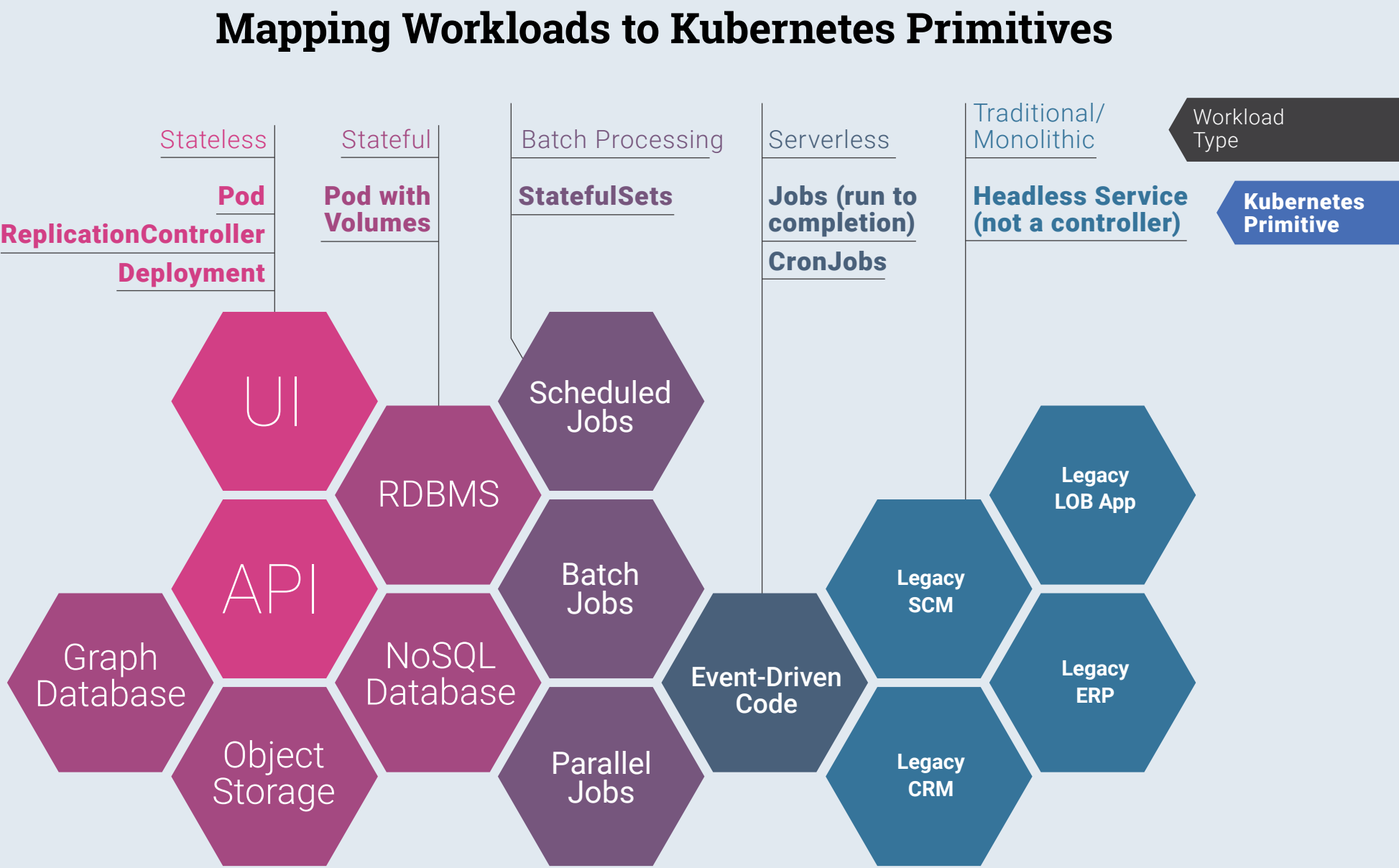
The depiction below (Fig. 2.2) maps various layers to the cloud-native application stack with Kubernetes primitives.

Best Practices for Deploying Cloud-Native Applications in Kubernetes

Understanding the mapping between cloud-native applications and Kubernetes primitives is important for DevOps engineers. But it’s equally important to follow some recognized best practices in deploying those workloads.

Here are 10 best practices for deploying and running cloud-native

FIG 2.2: DevOps engineers can define the desired configuration state through a declarative approach — each workload maps to a controller.



applications in Kubernetes. These techniques help DevOps teams in getting the optimal performance for their application deployments on Kubernetes.

1. Never Deploy “Naked” Pods. Naked pods are pods that are not a part of a ReplicaSet, ReplicationController or Deployment. Since it is easy, a common practice for both developers and operations teams is to package a container as a simple pod and deploy it in Kubernetes. Naked pods suffer from single points of failure as Kubernetes will not be able to reschedule them when a node fails. Always package pods as a ReplicationController or a Deployment.

2. Choose an Appropriate Controller for the Workload. Kubernetes has a variety of controllers mapped to specific workloads. Choose from ReplicationController, Deployment, StatefulSet, DaemonSet and Job controllers depending on the kind of workload.

3. Use an Init Container to Ensure That the Application is Initialized. A pod is a collection of one or more containers. An Init container in a pod runs before the application containers are started. Init containers are similar to regular containers but they always run to completion, and they run in a sequence. If a pod has multiple Init containers, each one must complete successfully before the next one is started.

Init containers are used to populate database tables, download files from remote locations to local volumes and check for other pods to become available. Avoid using a sidecar container when targeting Kubernetes. Sidecar containers packaged in the same pod are treated as normal containers by Kubernetes. Unlike Init containers, they start along with other containers of the pod, making it difficult to ensure that the initialization is done before starting the rest of the pods.

4. Launch Services Before the Workloads. Create a service before creating its corresponding backend workloads, such as Deployments or ReplicaSets, and before any workloads that need to access it. When Kubernetes starts a container, it provides environment variables pointing to all the services which were running when the container was started. When a pod is scheduled, it automatically gets populated with the environment variables of services created within the same namespace.

5. Use Deployment History to Rollback and Rollforward Versions.

One of the advantages of using Deployments over naked pods and ReplicaSets is the ability to rollback and rollforward versions. Always use Deployment Record to perform easier rollbacks. This capability mimics the ease of dealing with a Platform as a Service (PaaS), like Heroku or Engine Yard for deployments.

6. Use ConfigMaps and Secrets. Secrets securely store sensitive information, such as passwords, OAuth tokens and secure shell (SSH) keys. Moving this information to a secret is much safer and more flexible than embedding it in a pod definition or baking it into the container image.

ConfigMaps allows developers to decouple configuration artifacts from image content to keep containerized applications portable. They are a good replacement to hard-wired configuration supplied through external files.

7. Add Readiness Probe and Liveness Probe to Pods. The Kubernetes control plane relies on a readiness probe to know when a container is ready to start accepting traffic. A pod is considered ready when all of its containers are ready. One use of this signal is to control which pods are used as backends for services. When a pod is not ready, it is removed from service load balancers.

Kubernetes depends on the liveness probe to decide when to restart a container. When the liveness probe encounters a situation like a deadlock, where an application is running but unable to make progress, Kubernetes restarts the container to reset the state.

8. Define CPU and Memory Resource Limits to Containers and

Pods. The Pod specification allows resource limits to be specified for each container. A container is guaranteed to have as much memory as it requests, but it is not allowed to use more memory than the defined limit. This is also the case for CPU limits. The CPU resource is measured in CPU units, which typically translates to the virtual CPU (vCPU) or a core of the associated infrastructure.

9. Define Multiple Namespaces to Limit the Default Visibility of

Scope of Services. Kubernetes supports multiple virtual clusters backed by the same physical cluster called namespaces. For each service created in a Kubernetes cluster, there is a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container just uses `<service-name>`, it will resolve to the service which is local to a namespace. This is helpful for using the same configuration across multiple namespaces such as development, staging and production. To reach services from other namespaces, a fully qualified domain name (FQDN) may be used.

10. Configure Horizontal Pod Autoscaling for Dynamic Scaling of

Stateless Workloads. The Horizontal Pod Autoscaler (HPA) automatically scales the number of pods in a ReplicationController, Deployment or ReplicaSet based on observed CPU utilization or with custom metrics support. HPA is managed by the Kubernetes control plane. The resource consumption influences the behavior of the controller. The controller periodically adjusts the number of replicas in

a ReplicationController or Deployment to match the observed average CPU utilization to the target specified by a user. This configuration delivers optimal performance of stateless services.

Conclusion

One of reasons that Kubernetes has become successful is the flexibility and control it offers to developers and operators. Developers stay focused on shipping microservices without worrying about the deployment environment. DevOps engineers take the software, map the layers to appropriate primitives and deploy it in Kubernetes. This workflow and decoupling of software design and deployment is what makes Kubernetes unique.

This chapter focused on attributes of cloud-native applications, mapping cloud-native workload types to Kubernetes primitives and best practices for deploying and running applications in Kubernetes. In the next chapter we will explore to how to build CI/CD pipelines that automate deployments.

IMPROVE SECURITY WITH AUTOMATED IMAGE SCANNING THROUGH CI/CD



Automation through a CI/CD pipeline is key to securing an application deployed on Kubernetes. Using cloud-native security tools that hook right into Jenkins or your favorite CI/CD tool, enterprise

security teams can set policies for developers who are building container images. The pipeline enforces those policies through automated vulnerability scanning of each image during the build process. Developers only deploy images that the security team is confident in because they've been scanned.

“CI/CD automation is key because of the scale,” said [Liz Rice](#), technology evangelist at Aqua Security. “You couldn’t possibly manually check all these different images when you’re shipping potentially hundreds or thousands of deploys in a day.”

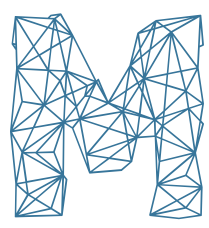
In this podcast, learn about how distributed architectures change the enterprise approach to security, how automation can improve security at scale and how Aqua’s tools — free as well as commercial enterprise versions — can help improve security through the entire application life cycle. [Listen on SoundCloud »](#)



Liz Rice is the Technology Evangelist with container security specialists [Aqua Security](#), where she also works on container-related open source projects including [kube-bench](#) and [manifesto](#). This year she is co-chair of the CNCF’s [KubeCon + CloudNativeCon](#) events taking place in Copenhagen, Shanghai and Seattle.

CONTINUOUS DELIVERY WITH SPINNAKER

by **CRAIG MARTIN**

 Modern application architecture releases should be frequent, fast and, above all, boring. This makes everyone happy — the developers, the business managers and the client. To this end, the growing tech movement to organize software teams and technologies around the notions of DevOps has created great interest in continuous delivery (CD) platforms.

DevOps, itself, promises golden opportunities: faster deployments for a marketplace that demands new features, greater accuracy of deployments through automation, faster feedback from clients through iteration, and even higher job satisfaction as teams cooperate to get features and fixes out the door faster. DevOps is a journey and not a destination. It means building cross-functional teams with common goals, aligning the organization around the architecture — [reversing Conway's Law](#) — and creating a culture of continuous improvement. One of the higher-level achievements in a DevOps journey is continuous delivery.

ThoughtWorks encapsulates the [ideal CD mindset](#). Here's how they describe it:

“*Continuous Delivery is the natural extension of Continuous Integration, an approach in which teams ensure that every change to the system is releasable, and release any version with the push of a button. Continuous Delivery aims to make releases boring, so that we can deliver frequently and get quick feedback on what users care about.*”

Implementing Kubernetes on its own doesn't magically achieve CD for your organization. The features Kubernetes provides, however, in modularity, available tooling and immutable infrastructure certainly make CD much easier to put in place. Although Kubernetes will help you define a container deployment and manage instances, it leaves it up to you as to how you will automate those deployments into environments.

In this chapter, The New Stack asked [Craig Martin](#), senior vice president of engineering at Kenzan, to discuss Spinnaker and how it illustrates the way CI/CD practices are evolving for cloud-native architectures. Kenzan is a top contributor to the open source Spinnaker project and has built its own open source framework for Kubernetes deployments with it. Here, Martin draws on his experience working with organizations to enact digital transformations through building large-scale microservices applications on top of Kubernetes and Spinnaker to explain:

- A new approach to CI/CD that's emerging for cloud-native architectures.
- The emergence of Spinnaker as a CD tool based on “state” management.
- The benefits and drawbacks of this new tool, which has not yet gained widespread adoption.
- Best practices for setting up CI/CD pipelines with Spinnaker on Kubernetes.
- New strategies for CI/CD on a cloud-native stack.

Ultimately, how companies employ CD onto Kubernetes is very important to achieving the ideals of push-button, boring deployments. Development groups have commonly answered the question of CD with established tools they are familiar with, namely Jenkins. Some 45 percent of respondents in [The New Stack's Kubernetes survey](#) said they use Jenkins to deploy applications to Kubernetes. This makes perfect sense. Companies have historically solved the problems of continuous integration (CI) first, through versioning code and repeatable builds in Jenkins. It's only natural that they would extend the job-running CI capability of Jenkins to then solve the problem of automating deployments. CD-savvy developers are even becoming familiar with new Jenkins 2.0 pipelines. In 2.0, an included pipeline plugin allows you to create Groovy scripts to orchestrate a fine-grained control over build, test and deployment stages. Other plugins, such as Blue Ocean, allow you to visualize these CI/CD pipelines in Jenkins.

While these highly customizable Jenkins pipelines can be built out through complex scripting, it still begs the question as to whether Jenkins is the right platform for the job of CD, or what platform, in fact, is the best tool for Kubernetes deployments on the cloud.

Enter Spinnaker

CD differs from CI. CI is a mechanism to merge and test code changes on an ongoing basis, often achieved by a tool like Jenkins. CD is the attempt to speed up and automate deployments, where an operator can push out multiple deployments in a week across numerous services, and know the exact condition of the applications and infrastructure in the course of the deployments. What is truly required for continuous delivery which is not provided by CI tools is a “state” machine. Such a state machine will have the ability to take an environment from one state to the next until it makes

it all the way to production. The machine will move the environment, such as Docker containers, through to production in an automated fashion, and will even have the ability to do things like rollbacks, canary deployments and scaling instances. This allows for the agile, push-button, automated deployments that an ideal CD mindset drives towards.

[Spinnaker](#) provides such a state machine. Spinnaker is an open source continuous delivery platform developed by Netflix to handle CD operations at high scale over its cloud network. It is a cloud-native pipeline management tool that supports integrations into all the major cloud providers, namely Amazon Web Services (AWS), Azure, Google Cloud Platform and OpenStack. It natively supports Kubernetes deployments.

Spinnaker is different than most other CI tools. CI technologies build code and run tests against it. CD technologies focus more on displaying the current state of the environment. CD takes software delivery a step further by automatically testing the software and pushing it into production using techniques such as canary testing and blue-green testing.

Spinnaker does not replace CI tools. It works alongside the tried-and-true workhorse of Jenkins: A Jenkins job can still handle building and storing artifacts for the CI portion, and once finished, the job can trigger a Spinnaker pipeline that deploys the application onto Kubernetes — or anywhere — for the CD portion.

We've worked with several companies to implement Jenkins side-by-side with Spinnaker, and have, at times, received furrowed-brow, questioning looks at the initial recommendation. Why go to all the trouble of learning Spinnaker as a secondary CD tool and integrating with it?

Spinnaker Features

Using Jenkins alone to enable the fine-grained control of pipelines that can do things like automate testing, rollbacks, visualization and templated

reuse would take quite a bit of custom Groovy code using Jenkins 2.0 pipelines. And after all this work, you would still not have a true “state” management tool.

Spinnaker already holds mastery at this state management, and provides the mentioned pipeline features out of the box, going well beyond tools like Jenkins.

There are many features that give it this mastery:

- **Cloud-Native Delivery Tool:** Spinnaker is cloud native. It was built for the cloud and in the cloud. This means that all the benefits of the cloud — autoresiliency, autoprovisioning and autoscaling — are native to Spinnaker.
- **Microservices based:** Spinnaker follows microservice architectural patterns. Along with utilizing microservices, each component is built to solve the needs of its domain, utilizing domain-driven design. This allows Spinnaker a high level of modularity and extensibility to meet your specific needs.
- **Delivery and Infrastructure Visibility:** Spinnaker provides a user interface (UI) that allows you to view your infrastructure and to see exactly where your code is residing. This gives you views of your load balancers, regions, ingress IPs, etc. The importance of seeing code deployments and the infrastructure cannot be overstated, as it will simplify your view into deployments.
- **Conditional Pipelining:** This is the ability to have pipelines spin off when certain criteria are met, for example: infrastructure, load, regions and failure types. Although this can be achieved with other tooling, we find Spinnaker pipelines the easiest and most intuitive to configure with conditional logic.

- **Robust API:** Spinnaker was built to be a consumed application programming interface (API) and is very thorough, allowing a great deal of customization as your deployments require it.
- **Scaling Capabilities:** As a result of the microservices architectural patterns that Spinnaker uses (more details in the next section), it can achieve high levels of scalability. Each microservice composing Spinnaker is capable of being horizontally scaled and each component has quite a bit of resiliency built into it. If you have a need for high scalability, then Spinnaker is the gold standard.
- **Built-in Deployment Strategies:** Out of the box, Spinnaker provides many commonly used deployment strategies, including Highlander, red/black and rolling red/black. You can drop these into your pipelines and use them wherever you see fit. It allows an operator to roll back a failed deployment right within the Spinnaker UI.
- **Flexibility of Deployment Modalities:** Spinnaker isn't just for deploying applications. It is meant to encompass all aspects of your stack, including deployment of infrastructure components and configurations. In this way, you can deploy infrastructure components, and then the application.
- **Multi-Region:** Spinnaker natively supports delivery pipelines that can deploy across regions in parallel. It can even support a multiple-cloud setup.
- **Kubernetes and Container Aware:** The pipelines within Spinnaker integrate easily with Kubernetes container deployments. The UI can even be used to manage instances. You can scale deployed pods from the UI during a peak, or terminate them as needed.
- **Flexibility of Deployment Targets:** Spinnaker supports a variety of

deployment targets, including mainstream cloud and container platforms such as Amazon Web Services, Azure, Cloud Foundry, DC/OS, Google Compute Engine (GCE), Kubernetes and OpenStack. Irrespective of the target environment, the deployment experience is always consistent.

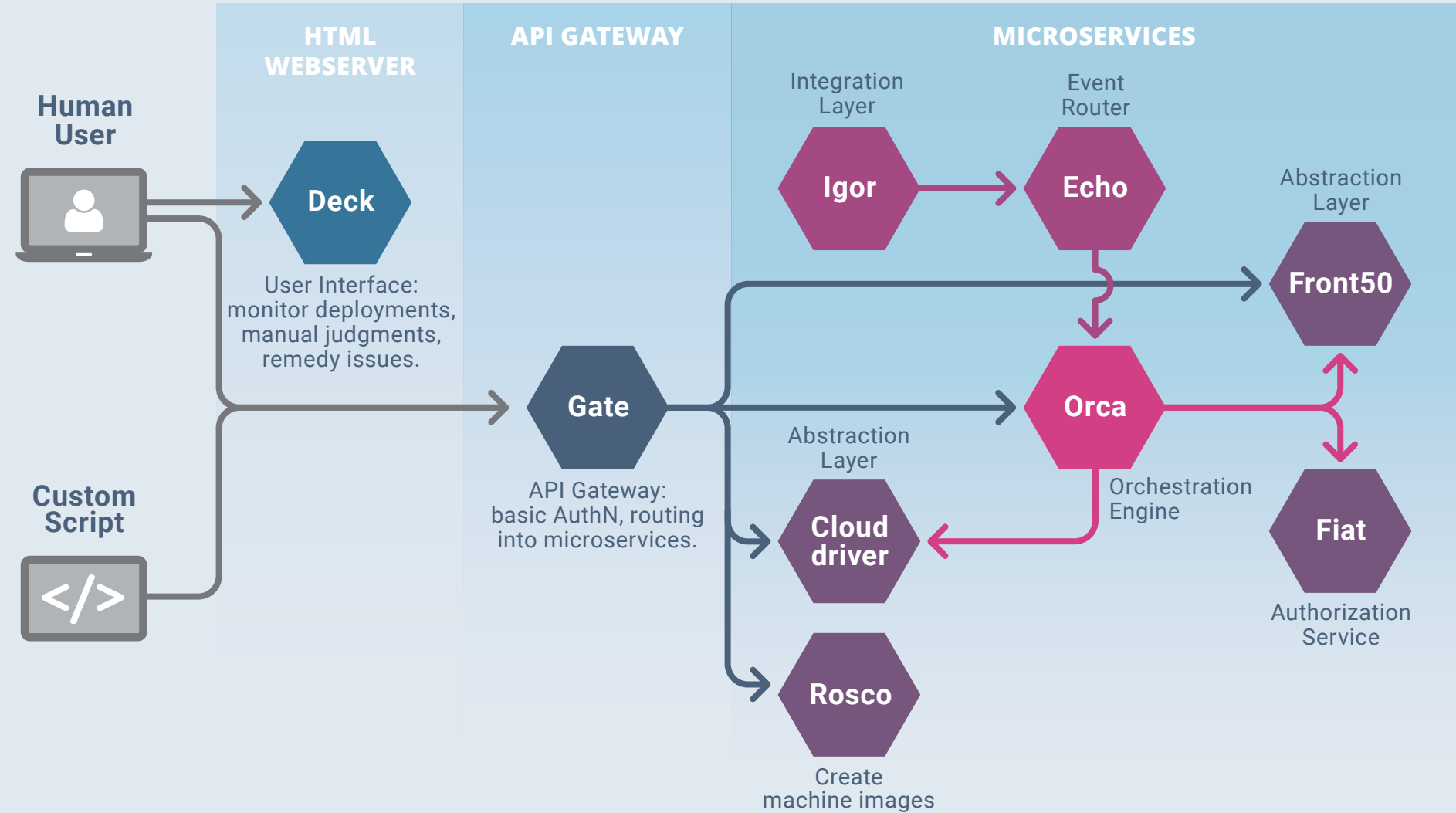
Spinnaker Architecture

Understanding the component architecture of Spinnaker is important to seeing its strengths. A typical Spinnaker installation includes a number of microservices that all work together to create and manage pipeline-based deployments.

Each component takes on specific roles and responsibilities within its domain.

FIG 3.1: Each component in Spinnaker’s modular architecture has its own responsibilities.

Spinnaker Components and Their Roles



- [Deck](#): A user interface to monitor the deployments, make manual judgements and remedy issues.
- [Gate](#): The API Gateway that is used to perform basic AuthN and routing into the microservices.
- [Orca](#): The orchestration engine that is used to actually perform the pipeline task and perform the work. It uses [Redis](#) to persist data while it is executing.
- [Clouddriver](#): The abstraction layer that Spinnaker uses to communicate with cloud providers. Also stores local caches of deployed resources.
- [Front50](#): The abstraction layer for persisting metadata, config, notifications, etc. By default it uses Cassandra, but many different datastores can be used.
- [Rosco](#): This is the microservice that is used to create machine images, for example: AWS Amazon Machine Image (AMI), Azure virtual machine (VM) images, and Google Compute Engine (GCE) images.
- [Igor](#): An integration layer in your continuous integration tooling, such as Jenkins and Travis; and Git repositories, such as Bitbucket, GitHub and Stash.
- [Echo](#): This is an event router that is used to communicate all critical events to the properly configured listeners, such as Slack and short message service (SMS).
- [Fiat](#): Spinnaker's authorization service. It handles the authentication and authorization for users, applications and service accounts.
- [Halyard](#): Spinnaker's configuration service that is used to install, maintain and upgrade Spinnaker itself. Halyard is an important

“Terraform-like” tool, creating a golden deployment configuration for Spinnaker that can be used for recovery or spawning additional instances.

Spinnaker has a smart component architecture where each microservice domain has its own API and can extend in a modular way. This allows for greater flexibility, more features and future integrations with any number of technologies.

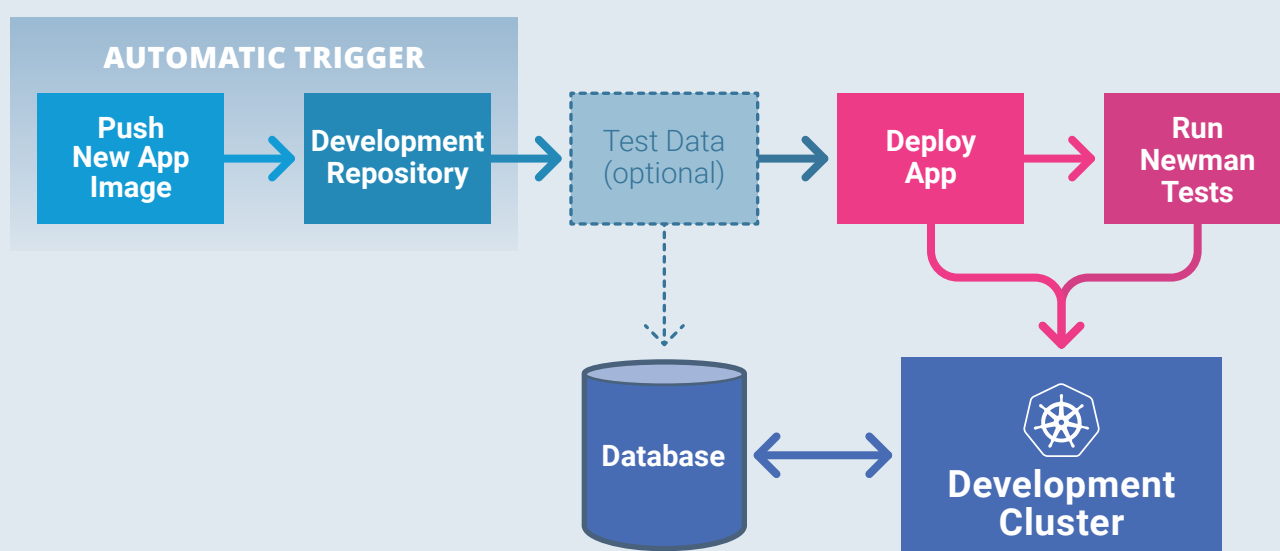
The Beauty of Pipelines

While we’ve mentioned the idea of pipelines in Spinnaker, we haven’t yet described what they do or seen them in action. Pipelines are at the core of CD capabilities, by orchestrating a repeatable deployment over stages.

Each pipeline in Spinnaker starts with a specific trigger — this could be a Git commit, a manual start, a Jenkins build, a push to a Docker repository or another conditional trigger. The pipeline proceeds through several user-defined stages. Stages could involve the actual deployment, running automated tests for the specific environment, or even rolling back a deployment if smoke tests fail. What you want to accomplish depends on

FIG 3.2: A new image in the development repository triggers the Spinnaker development (dev) pipeline to deploy the application to a Kubernetes dev cluster.

Spinnaker Development Pipeline



the environment, so individual pipelines are typically created per the environment you deploy in.

Development Environment

For example, in a development (dev) environment, you might set up a very simple Spinnaker deployment which runs often. Triggering off a new image being pushed to the container image repository, it deploys the application to the dev Kubernetes cluster, then on post-deploy runs a number of curls against the application's REST API using a Newman script.

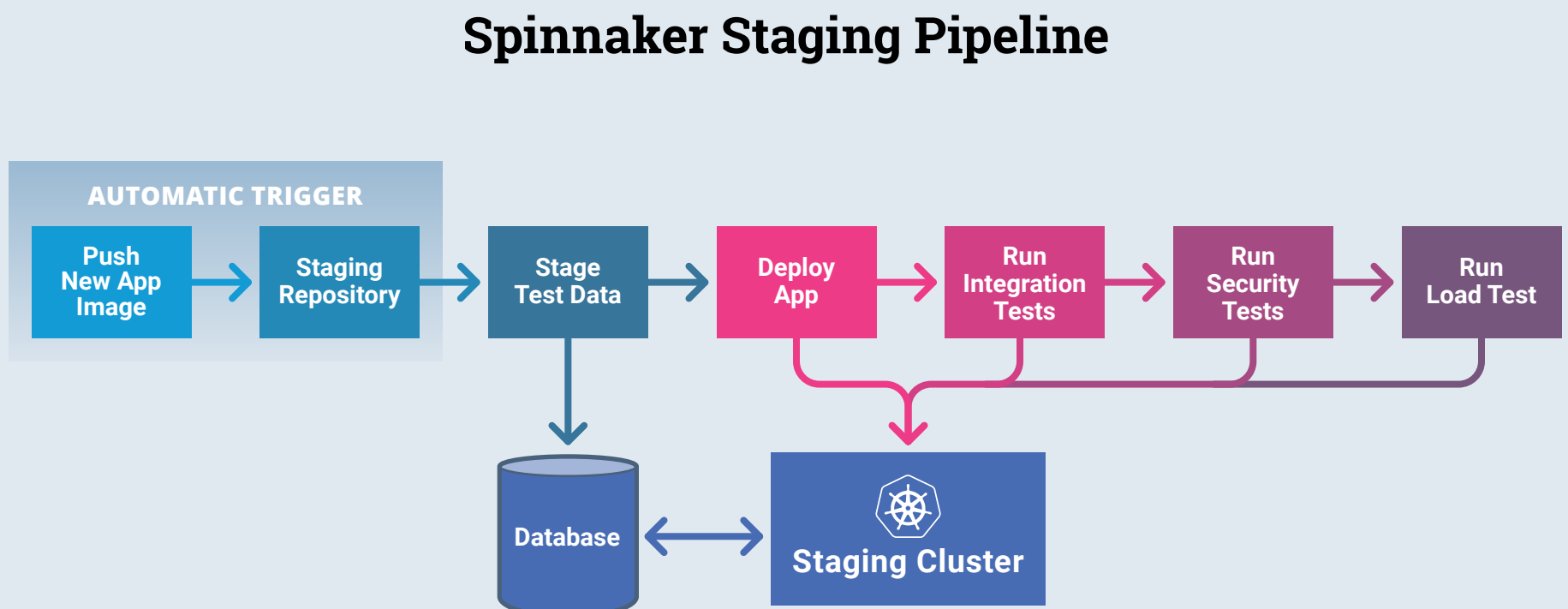
Staging Environment

In the staging environment, you might set up another pipeline that is a bit more complex. Triggering off a new image pushed to your staging repository, you first create some new test data using a database insert. The next few stages deploy the application, run integration tests utilizing the test data, perform security penetration tests and finally load tests to simulate peaks.

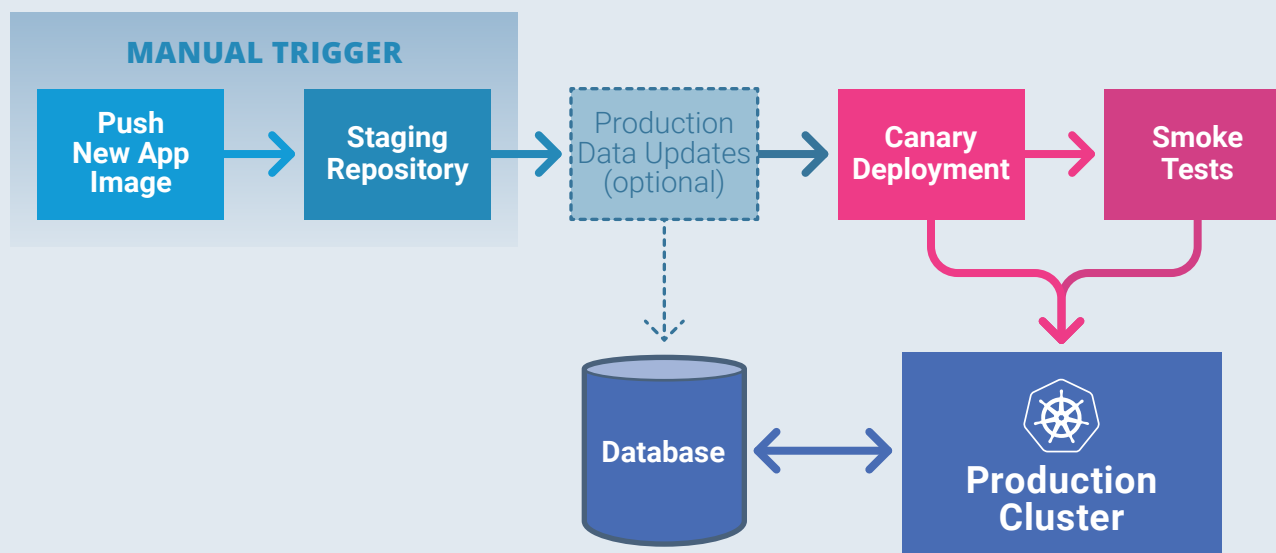
Production Environment

In the production (prod) environment, instead of triggering off of Jenkins, we allow operations to start the pipeline manually. Using Spinnaker's

FIG 3.3: A new image in the staging repository triggers the Spinnaker staging pipeline to deploy the application to a staging cluster.



Spinnaker Production Pipeline



Source: Kenzan

© 2018 THE NEW STACK

FIG 3.4: The operations team manually triggers the prod pipeline to deploy an application to production on Kubernetes.

built-in capabilities, we've set up a canary deployment which gradually routes requests to the new version of the application, and also includes a stage that performs smoke tests for good measure. For more on canary deployments, see [best practices](#).

Throughout these various pipelines, Spinnaker's infrastructure view provides visibility regarding what did and did not happen along the pipelines, what tests passed or failed and what versions were deployed. Having a CD orchestration tool that provides an applications-on-infrastructure view makes supporting the running application that much easier for operations, giving a level of insight and control that creates push-button deployments across environments.

All of the above pipelines were constructed inside the same Spinnaker UI. Although scripting pipelines is possible for more complex operations, the templates in the Spinnaker UI make stringing together pipeline stages fairly easy, with most tools and options needed for robust deployments. In our experience, pipelines and its concept of Pipeline Templates are one of the areas where Spinnaker really shines. Creating custom [pipeline templates](#) is an easy way to create reusable pipeline modules that can be

applied in varying situations. We have found this to be a very useful way to give each team control over their pipelines, but ensure that individual components are only built once and built correctly. The templates make reuse natural, and not a chore requiring best practices or lengthy documentation, which custom code templates might require.

And while the above pipelines demonstrate what deployments might look like across typical development, staging and production environments, the ultimate focus of CD on fast deployments should cause you to question having so many environments. Could all of the above stages be accomplished in fewer environments? Deployment strategies, like canary, combined with the automation and accuracy Spinnaker provides in its pipeline stages, could very well provide a path towards having code pass through fewer environments, making it to production in fewer steps.

Example Implementations

It's helpful to look at a few actual implementations to see the flexibility and power of Spinnaker. At Kenzan, we've helped implement many of the following Spinnaker use cases with clients.

Case 1: All in with the IaaS

For organizations whose journey to the cloud is not burdened by supporting legacy infrastructure and applications, they may choose to leverage Infrastructure as a Service (IaaS) capabilities for building applications. The motivation is that IaaS now provides various classic continuous integration capabilities, allowing organizations to use their services rather than implementing specific build tools. For example, [Google Cloud Container Builder](#) — which is also used to create the Spinnaker container images for release — is a Google Cloud service that can be employed to perform various build and low-level integration actions.

Spinnaker integrates with both Google Cloud and Kubernetes. A container image build in Google Cloud Container Builder can act as a pipeline trigger into Spinnaker, which can then deploy your containers on a Kubernetes cluster. An organization that uses such a cloud-first, pro-iaaS approach can quickly build and deploy containerized applications with all the appropriate systems development life cycle (SDLC) capabilities. This provides the organization with the advantage of not having to commit to several separate and distinct SDLC tools. Additionally, flexibility is still retained through the existing custom Spinnaker stages, which allows integrations with various SDLC tools.

The outcome of the approach is that the organization can then spend more time supporting applications and less time on app tooling.

Case 2: Spawning Developer Environments

Spinnaker is able to deploy both code and infrastructure. Starting from zero, it can deploy and manage an entire stack: a functional environment that includes Spinnaker itself.

To demonstrate this capability, let's assume I am an infrastructure-as-code, microservices and containerization-embracing company. My e-commerce platform is composed of 100 different services with various instances of those services. I would like to enable the following SDLC conditions and practices:

1. I would like to have developers who are working on features be able to obtain a duplicate of the last known good (LKG) of the entire platform environment, except for the service they are enhancing.
2. I would like to have disposable integration environments.
3. I would like to spawn disposable performance and pen testing environments.

These are goals that many organizations dream of, but are often incapable of executing due to the complexity of replicating environments and the cost of doing so.

Spinnaker allows an elegant solution to a potentially complex problem by being both infrastructure-aware and having the ability to replicate itself. A solution we've implemented is having an "Environment Creator Spinnaker" instance that creates new environments and infrastructure for the developer. It starts by deploying Kubernetes. Using a predefined Halyard definition, it then deploys a new Spinnaker instance onto the Kubernetes cluster. Then the pipeline definitions are run in the appropriate sequence to redeploy the desired version of all the applications and their infrastructure. Spinnaker deploys Kubernetes and Spinnaker, running pipelines to create a complete, disposable environment. In this way, separate environments can be spun up for development, integration testing and performance testing.

You may be thinking, that's great, but what about the cost of such environments? You could unknowingly spin up a number of cloud resources and leave them in place, only to find out later your usage bill has tripled. To help solve this, a manual decision stage can be put at the end of the Environment Creator pipeline, creating a manual pause that allows adequate time for testing to occur, then finishing the pipeline by destroying the entire environment. While there is no way to escape the full cost of creating such disposable environments, Spinnaker can help out by mitigating their time to live.

Case 3: Database Migrations with Spinnaker

Spinnaker is more than just a tool for deploying applications. It can be used to automate infrastructure-related tasks. For example, using Spinnaker pipelines you can accomplish the painfully difficult task of schema migrations — often just called database migrations. Most

organizations perform database migrations as a pseudo-manual process. Some may attempt to include a schema change as part of application start up. However, when you have a running application attempting a canary release, this could lead to indeterminate data representation. As a result, database migrations are considered curious special cases that often lead to “one off” deployments.

Spinnaker solves this by [enabling database migrations](#) to run directly in a release pipeline. Before the deploy stage for your application, you can employ a stage to perform the data migration task, something a database administrator (DBA) would typically have to perform. You can even include a “check schema for changes” step in your deployment pipeline on every release. This makes the obscure one-off task that is operationally expensive just as cheap and easy as deploying a new container image. Automating database migrations this way also provides a level of repeatable reliability; the alternative — custom scripts or manual steps — are typically much more brittle.

Spinnaker as the Standard

We often get the question as to whether Spinnaker is going to emerge as the standard CD tool going forward. It is hard to predict, but that hasn’t stopped us from having an opinion based on experience and the general landscape we see in software development. The short answer is that we believe it will emerge as the preeminent continuous delivery tool, though it will take some time.

Spinnaker already has the backing of some large, forward-thinking companies which are top contributors to the project, including Google, Microsoft, Netflix, Oracle and Pivotal. And it is one of the first CD tools built from the start to allow DevOps teams to truly take advantage of cloud-native architectures to improve efficiency, speed and agility. But it

is still early days. We have seen other tools begin to evolve to include cloud-native CD capabilities and more will follow. In addition, any contender in the space will need to address the need to integrate with, or incorporate, build tooling to allow for complete application lifecycle management. Spinnaker is the closest we've seen that delivers the scaling capabilities and features needed in a cloud-native, continuous delivery tool but much development work is still needed to get it to a turnkey, enterprise-ready state.

A Cultural Shift Towards Delivery

Most companies are just getting started on their journey towards transforming into a DevOps-based organization. For these companies, delivery is still not seen as a true priority or a business differentiator. At Kenzan, we push the notion that speed and accuracy of deployments should be just as important for a product as any actual feature. Companies are generally slow to adopt this mentality.

Focusing on deployments is a cultural shift for companies that involves organizational change. It involves changing team structures, changing processes and changing culture, all of which take a longer period to root and grow. And while companies may be aware of the ability to automate deployments through CI tools like Jenkins, they may initially overlook a long-term strategic plan for achieving faster, automated deployments and the benefits a fully-featured CD tool like Spinnaker can provide. Because of this, adopting a truly deployment-focused platform like Spinnaker may not be realized for some time.

A Complexity Shift in Tooling

On the technology side of the equation, the initial versions of Spinnaker proved overly complex and involved an elaborate setup. The Spinnaker team is now working very hard to simplify the setup of Spinnaker, but the early versions did make it less approachable for the community to adopt.

Fortunately, the current Kubernetes installation and setup via [Helm](#) is probably the easiest method yet to configure and set up Spinnaker. This has made it much more approachable for those integrating with Kubernetes, yet the general complexity of Spinnaker remains a boundary for many organizations.

In general, we've gradually seen complexity shifting away from building and assembly of code towards orchestration of releases. Build tools are starting to commoditize and become much simpler, such as Travis CI and Jenkins. This commoditization has been driven by the simplification of using immutable infrastructures, particularly Docker containers, where building can happen directly in the container environment that it will run in, and the entire container is then promoted through the pipeline. As more and more organizations get comfortable with building custom code using containers and other immutable constructs, they will spend less cycles on building that code and shift into solving the problems of orchestrated releases. We are helping many of our customers see an "orchestration-first" mindset when thinking about software. When approaching a new development, we start by considering how it will be deployed and automated before we consider how it is built or even the architecture within the features. From this viewpoint, building code becomes just another implementation decision. We eventually foresee most organizations taking this view, and once the shift takes place, Spinnaker will be well-suited for the new wave of users who will require the orchestration-native capabilities that Spinnaker offers. There are other tools out there, such as Jenkins Pipelines, that can do basic orchestration, but they don't scale or have nearly the capability of Spinnaker.

Standing Out in a Crowd

Overall, Spinnaker has jumped into what might appear to be a crowded CI/CD marketplace. It includes a number of alternatives, alongside tools

like the aforementioned Jenkins 2.0. Most of the solutions are CI tools that have been extended with CD capability, with a few others that focus on CD only.

- [AWS CodePipeline](#) is a CD service built specifically for AWS. It allows the creation of pipelines for building and testing code with AWS CodeBuild and then deploying applications. It also integrates with AWS CloudFormation, allowing you to deploy complete application stacks. It provides a single UI for deployments, but you need to use other AWS features to get full visibility.
- [CircleCI](#) is a CI-centered tool that has expanded to pipeline capabilities. It is available hosted and self-hosted. Its greatest capabilities reside in not trying to mimic the architecture of legacy CI tools like Jenkins. Being able to support a hosted configuration but still isolate build runtimes make CircleCI an enterprise-grade solution where runtime isolation is important. In addition to this, CircleCI makes getting started with CI/CD easy.
- [GitLab CI](#) is the leading GitHub clone that an organization can deploy themselves. If you are looking for a self-hosted Git and are not opting for [Bitbucket](#), GitLab is a good choice, particularly with the CI/CD capabilities that GitLab 8+ provides. As the name suggests, GitLab CI is more about CI, but one can use GitLab's pipeline capabilities for CD tasks. With these pipelines one can sequence a code build, a Docker build and deploy the image to Kubernetes. However, these are all file-based configurations that require familiarity with Kubernetes manifests.
- [Harness](#) is a Software as a Service (SaaS) continuous delivery tool that can integrate into your custom environments. Recent additions to Harness have allowed it to start leveraging artificial intelligence

(AI) to predict and see anomalies in your environments. The benefit of SaaS is it will abstract away the complexity of establishing your own CD tooling, but you could be limited in customizations to the specific tools.

- [Jenkins X](#) is a CI/CD solution based on the backbone of Jenkins that is new as of March 2018. It is designed to specifically build and deploy containers onto Kubernetes in the cloud. It attempts to simplify the process of CI/CD by automatically producing a number of predefined things for you: Jenkinsfiles, Dockerfiles, Helm Charts, Kubernetes Clusters, namespaces and even environments. It also uses predefined automation to trigger builds and deploys from Git commits. Jenkins X is still in its infancy as a tool, and currently only runs from the command line; it does not yet have a UI for managing deployments.

The above tools include the most popular and relevant alternatives on the market; there are a number of other CI/CD solutions that we have not mentioned. What is important to realize about tools like CircleCI, Gitlab CI, and Jenkins X is that they are primarily CI solutions that are now being retrofitted to do what Spinnaker has been built to do from the beginning: Focus on cloud-based deployments. While retrofitted CI tools will likely provide a base level of capability for deployments, they may not provide the scalability, extensibility and community support that Spinnaker has proven itself with. And most importantly, with a focus on CI, it may take a long time before these tools match Spinnaker's capability as a true state machine: allowing an operator to automate changes, perform canary deployments, roll them back and scale instances, all while having a high-level view of which applications and infrastructure are in place. When examining the CD capability of any tool, it is important to remember the ideal CD goal of having a state machine that enables fast deployments for your organization.

Build Tooling as Part of Spinnaker

It is possible that the Spinnaker community will attempt to roll its own CI build tool that will sit alongside or be a part of Spinnaker. Whether it replaces Jenkins or other tools is not as important as the opportunities it would create. Using two tools to do the build and orchestration is often a barrier for adopting Spinnaker, and people end up reverting to what they know in tools like Jenkins. If a CI solution is built into Spinnaker, it would open the door to doing the entire build and orchestration process with one set of tooling.

Spinnaker Best Practices

Over the years, we've picked up a number of lessons learned that we try to leverage for engagements where we are moving clients to a CD pipeline with Spinnaker and Kubernetes.

1. Ensure Resilience

It is important to put as much emphasis in ensuring resiliency for your Spinnaker stack as you would for any application in your infrastructure. Ensuring uptime of your deployment tooling is as important as any feature or application. Fortunately, Spinnaker is already built with resiliency in mind, and it typically only requires configuration to achieve your specific needs. We suggest the following setup:

- **Multiregion:** If you have a multiregion setup, then ensure that each cluster has a full set of Spinnaker components.
- **Data Replication:** Depending on the data stores that you are using for your components — Cassandra and Redis — it is important to configure them to replicate across regions and datastores. We typically find that the out-of-the-box configuration for data replication, such as Active Active, Active Passive and Eventual Consistency, tends to meet

the needs of our installs.

- **Chaos Engineering:** Spinnaker integrates very nicely with chaos monkeys, and we find this very useful in our application pipelines to ensure that they are developed with the proper level of resilience.

We recommend performing chaos engineering and testing on your Spinnaker setup as well. This will show that it can handle failures so that you can be sure that your deployment pipelines will not be impacted by outages.

2. Employ Namespacing

We typically put our Spinnaker installation in a separate namespace within Kubernetes. This permits us the ability to size the Spinnaker resources at the entire namespace level, and also prevents against Spinnaker resources taking away from other namespaces or vice versa. Having partitioned Spinnaker off, we can then closely monitor the needed resources for our deployment microservices.

From within its own namespace, it should be noted that Spinnaker can deploy into other specific namespaces. This is a very nice ability that allows deployments to only target one namespace at a time.

3. Monitor Spinnaker

Monitoring Spinnaker is as important as monitoring any application in your infrastructure. Spinnaker currently supports three major monitoring systems — Datadog, Prometheus and Stackdriver — but others could be added relatively easily. We recommend using the same monitoring tools that you use for the rest of your infrastructure. The next chapter will go into more detail on monitoring with Prometheus.

- **Understanding Monitoring:** Spinnaker, by design, has very powerful monitoring capabilities, but understanding how it works is critical.

Each microservice is instrumented to capture and expose events and metrics, while separate daemons exist to gather this data and push it to the monitoring tool of choice. For a deeper dive, see [how monitoring works within Spinnaker](#).

- **Dashboards:** We typically use the supplied dashboards that are provided by our monitoring tool, such as Prometheus, but focus on two main views into Spinnaker. The first is the health of Spinnaker overall. This includes data from the clusters, nodes and namespaces running Spinnaker and the health of any integration points such as Jenkins. The second view is monitoring each microservice individually. We typically look for error rates and latency within the microservices.
- **Weigh Value Differential Over Rates:** A monitoring practice that we took from the Spinnaker team was to look for trends in data and to not care as much about the current rate or count. This has proven to be very good advice as the volatility in load is very high in a delivery platform.

4. Version Your ConfigMaps and Secrets

If you are using ConfigMaps and Secrets that are outside of your Docker containers, then it is very important to version them. Without versioning them, you will not be able to roll back in a Spinnaker pipeline should the configuration prove bad. It will likely result in a “roll forward” situation that will require a new build and new configurations.

5. Use Canary Deployments

While Spinnaker supports many deployment strategies within pipelines, Kenzan is keen on using the canary strategy. This allows us to slowly roll out the feature and test it directly in production. We typically use some form of automated canary analysis (ACA) to monitor the health of the deployment and compare logs files of the newly deployed to the previous

deployment. Fortunately, this is getting even easier to accomplish now that Google and Netflix have open sourced [Kayenta](#), which integrates with Spinnaker to monitor the quality of canary deployments. This was released in April of 2018 and will make achieving continuous delivery — and then continuous deployment — easier than ever.

6. Use Feature Flags

We tend towards using feature flags as a design practice for applications. This gives the ability to still deliver the feature when ready, but turn it on at a later date and not create a backlog of items waiting to be deployed via Spinnaker.

7. Use Pipeline Templates

Using [Spinnaker Pipeline Templates](#) allows you to standardize a set of pipeline components, but still give each team some flexibility to decide which parts they need and even the order of events. This is a very powerful way to ensure that reuse is happening across all of your pipelines.

Broader CD Best Practices

The move to using Spinnaker isn't simply about putting a piece of technology in place. It will likely go hand-in-hand with organizational shifts in thinking as well as enacting some key CD practices.

1. Mindset Shift

The overall organization needs to support CD. This means that the business, development and operations need to align their practices to support continuous delivery. Over the years, we have seen several common themes emerge that are key to this journey:

- **Focus on Releasing Code and Not Building Code:** This has been mentioned a couple of times but it is probably worth emphasizing

again. It is important that everyone sees quality and speed of delivery as the most important aspect of any application.

- **Fund as a Product and Not a Project:** Too many organizations are still funding and building budgets around projects. Each project can have many features, and sometimes these budgets won't allow features to be released independently. To ensure that releases are given priority, funds need to flow into a product as a whole.
- **Establish a Culture of Automation:** Your organization should support a culture of automation. This involves looking at new automation technologies, but also avoiding any infrastructure or application code that cannot be automated. This will kill the automation of your pipelines.
- **You Build it You Own it:** Creating ownership with the team that built the code is paramount in our experience. Every piece of code — microservice, pipeline and infrastructure — should have a clear owner, and we find it best not to create separations between the teams that manage and the teams that build the code. These types of separations typically end up creating handoffs that halt or impede the speed and automation CD pipelines are inherently designed for.

2. Infrastructure as Code

Automating everything is made much easier if your infrastructure is managed as code and configuration. This will make it much easier to create consistencies and parity across all of your environments.

Accomplishing this is dependent on your specific implementation, but we typically use some sort of scripting, such as with Terraform, and also employ dedicated infrastructure pipelines to automate the deployment. This ensures that individuals are never manually hand tweaking environments without the proper checks and balances in place.

3. Fail Early with the Testing Pyramid

Automated testing and failing early are important to speeding up delivery pipelines. At Kenzan, we subscribe heavily to the testing pyramid, where the bulk of tests occur at the base with unit and integration tests, and further up we use fewer end-to-end and UI tests for basic smoke testing. End-to-end and UI tests tend to be the slowest and most brittle. By running complex logic in the faster running unit tests and integration tests, this allows us to fail quickly and catch bugs earlier in the lower environments.

4. Consistent Branching Model

While Spinnaker is certainly capable of handling multiple Git branching models, we have found it easier to manage if all microservices and applications are using a consistent branching and versioning model across all of your code bases. Without a consistent branching model, you wind up adding different branch triggers in Spinnaker for different applications and potentially even different versioning and tagging. This burdens development with unnecessary complexity, in our experience.

5. Backwards Compatible Changes

Every change must be backwards compatible! Period. As soon as you start releasing features that are not backwards compatible, the rollback strategies and coupling becomes very difficult. This should never happen.

6. Start Small

You won't achieve full CD in one week. The process will take time to shift over all projects. When moving towards implementing CD in an organization, we typically start very small with a single application or group. This allows you to prove out the ideas and organizational changes on a smaller scale and then roll them out to a larger group.

Spinnaker Tutorial

We hope this chapter has piqued your interest in Spinnaker as a CD platform. If you'd like to get started with the tool, Kenzan has an open source repository that we use to set up a fully functioning CD environment using Spinnaker (with Jenkins) running within a Google Kubernetes Engine (GKE). The tool uses some simple Terraform scripts to set up and configure your environment, and is a great way to begin examining Spinnaker hands-on. Check it out at: <https://github.com/kenzanlabs/capstan>

A NEW APPROACH TO DEVOPS WITH SPINNAKER ON KUBERNETES



As organizations look to DevOps as a means to achieve digital transformation, they realize they must accelerate the end-to-end software delivery process, and also make it safer.

“It can seem like a boil-the-ocean type of problem. That makes it hard to figure out how you’re going to incrementally derive value from it,” said Andrew Phillips, a product manager in Google Cloud Platform’s DevOps division.

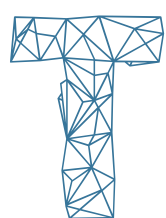
By separating the developer feedback cycle from the rollout process, organizations can find a manageable starting point. Tools like Spinnaker were built for this purpose, Phillips said, “to provide an abstraction so that development teams can have a simplified experience, while still providing the operations teams with the ability to manage and tweak and define it in exactly the way that makes the most sense for the organization.” [Listen on SoundCloud »](#)



Andrew Phillips is a project manager at [Google Cloud Platform](#) and a frequent contributor to the continuous delivery and DevOps space. He’s been a software engineer, team lead, infrastructure builder (a.k.a. head of duct tape) and community evangelist, and contributes to a number of open source projects. He is a regular speaker, author and co-organizer of ContainerDays Boston and NYC.

MONITORING IN THE CLOUD-NATIVE ERA

by **IAN CROSBY, MAARTEN HOOGENDOORN, THIJS SCHNITGER AND ETIENNE TREMEL**



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 [four pillars to observability](#):

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

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

Application Performance Management

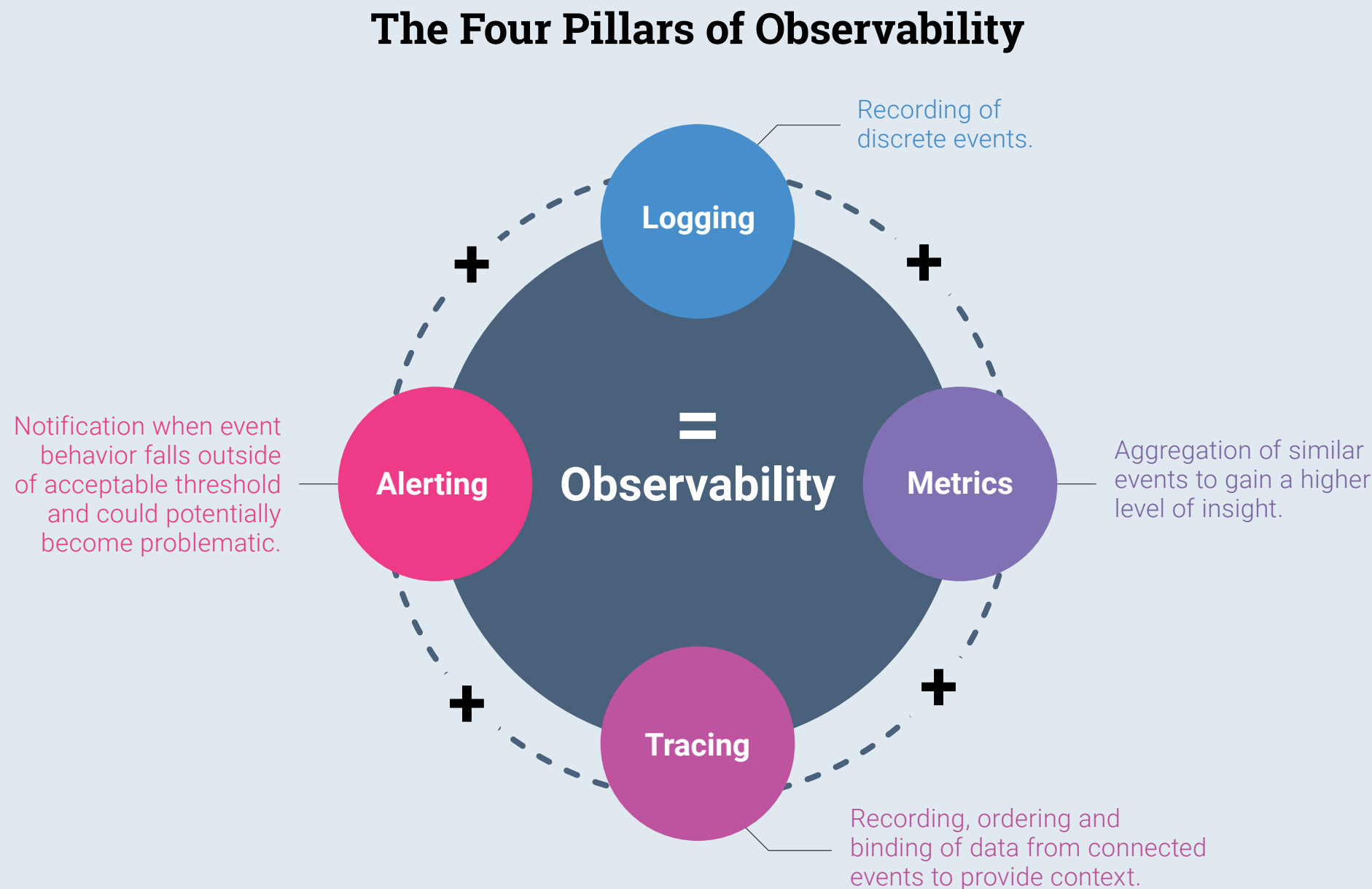
The dynamic nature of cloud-native systems poses new challenges for application performance management (APM). First of all, cloud-native systems are by definition more transient and complex than traditional systems. The components making up the system are no longer static, but ephemeral — appearing on 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 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 and 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

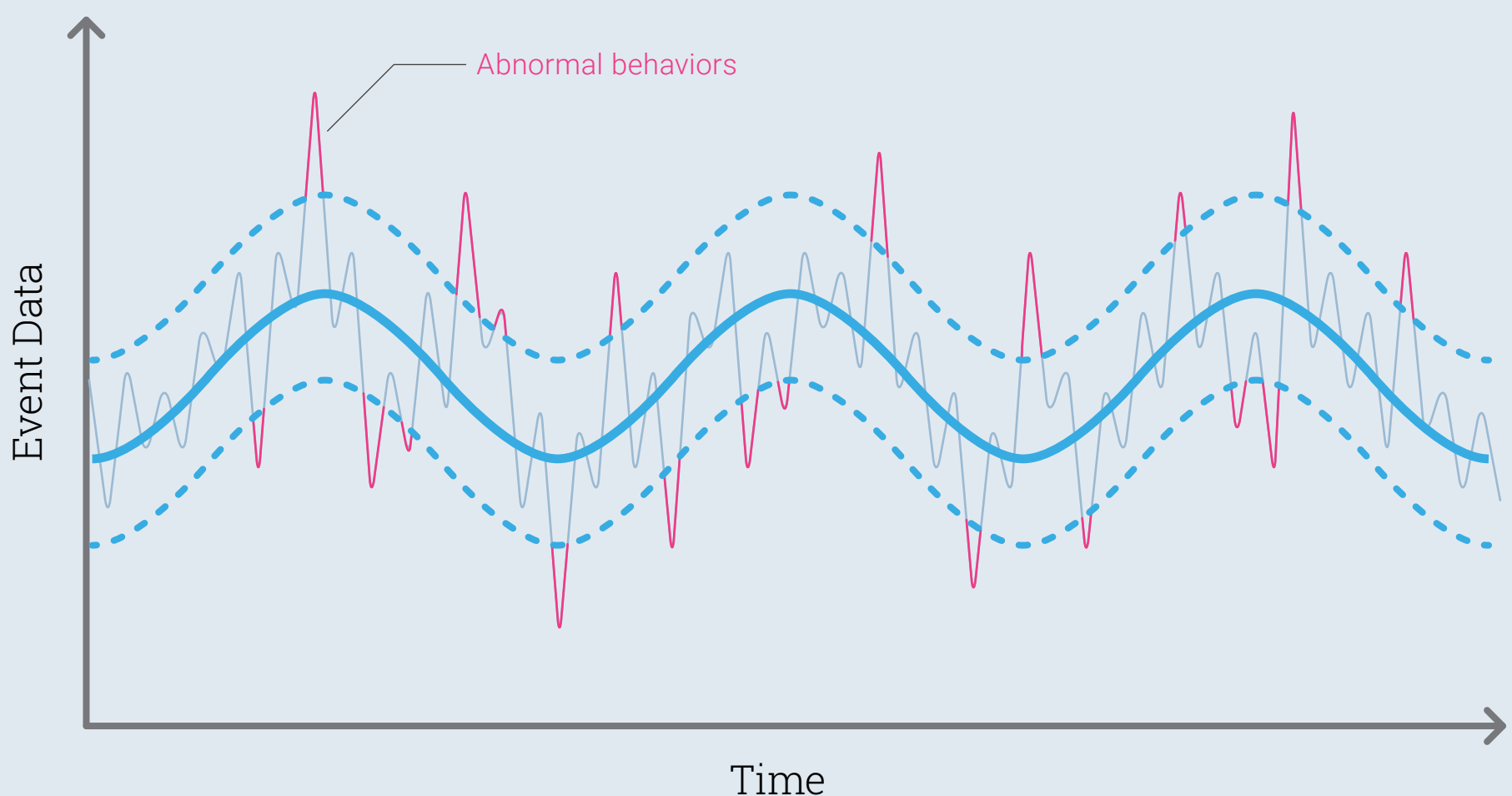
attempt to bring the system back to its normal state.

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 into the model. It is

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.*

The Holt-Winters Method of Anomaly Detection



one of the many methods that can be used to predict data points in a series over time. Figure 4.2 provides an overview of what the Holt-Winters method evaluates.

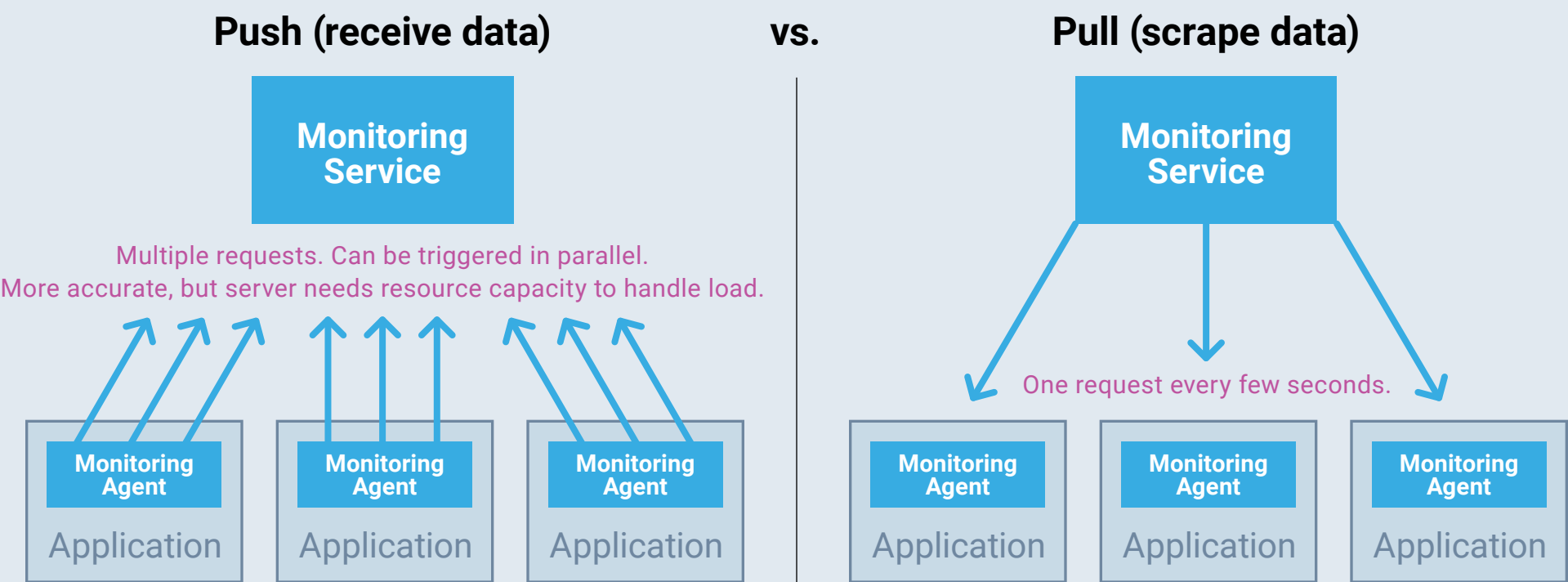
Push vs. Pull Data Collection

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 instance health is based on the event it receives. The push

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



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

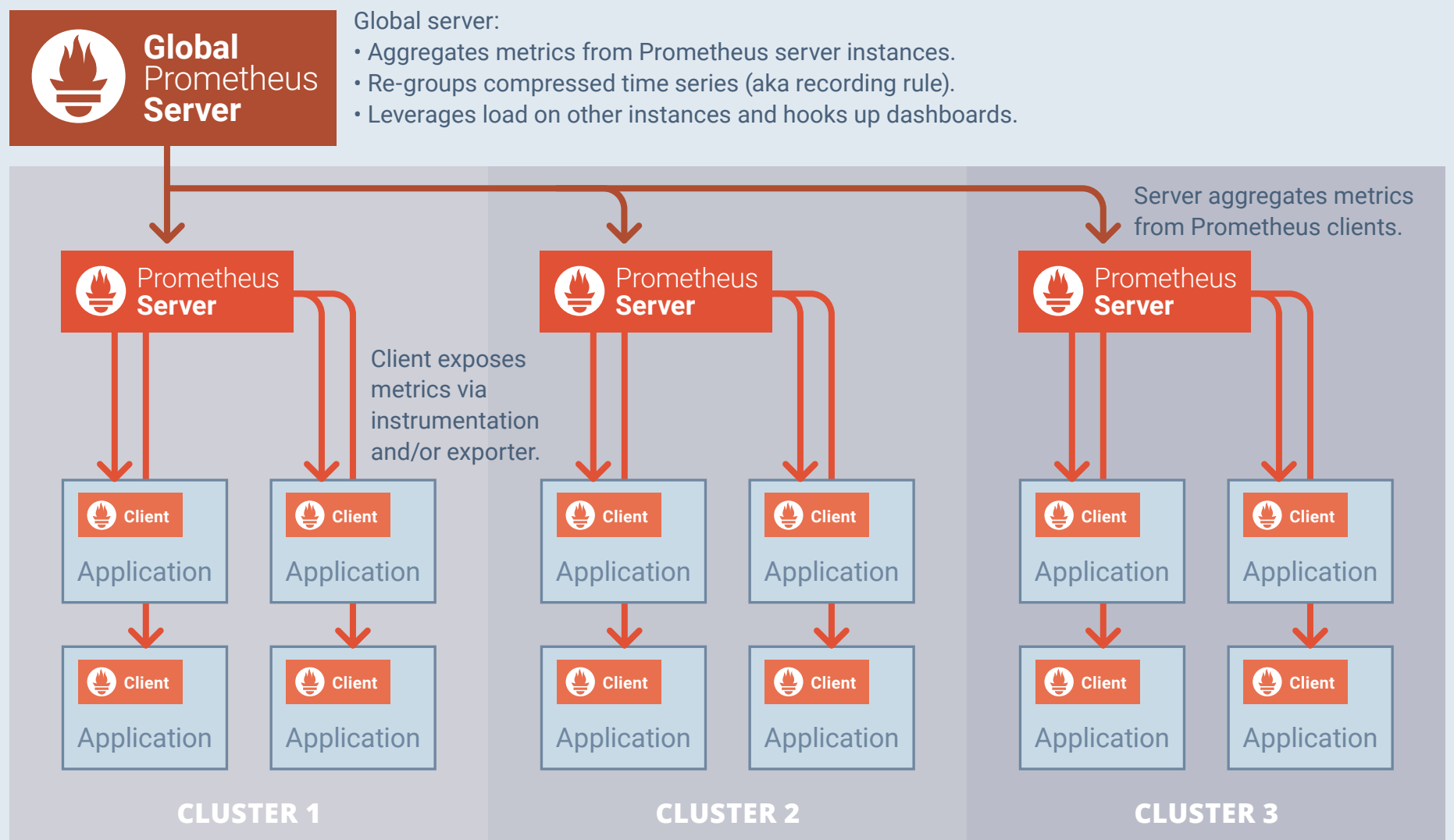
the load balancer can prevent such flickering of unsynchronised time series to be displayed on a dashboard.

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 coordination between the projects' communities.

Monitor as a Service, Not as a Machine



Source: <https://www.slideshare.net/brianbrazil/prometheus-overview>

© 2018 **THE NEW STACK**

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

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 multidimensional 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.

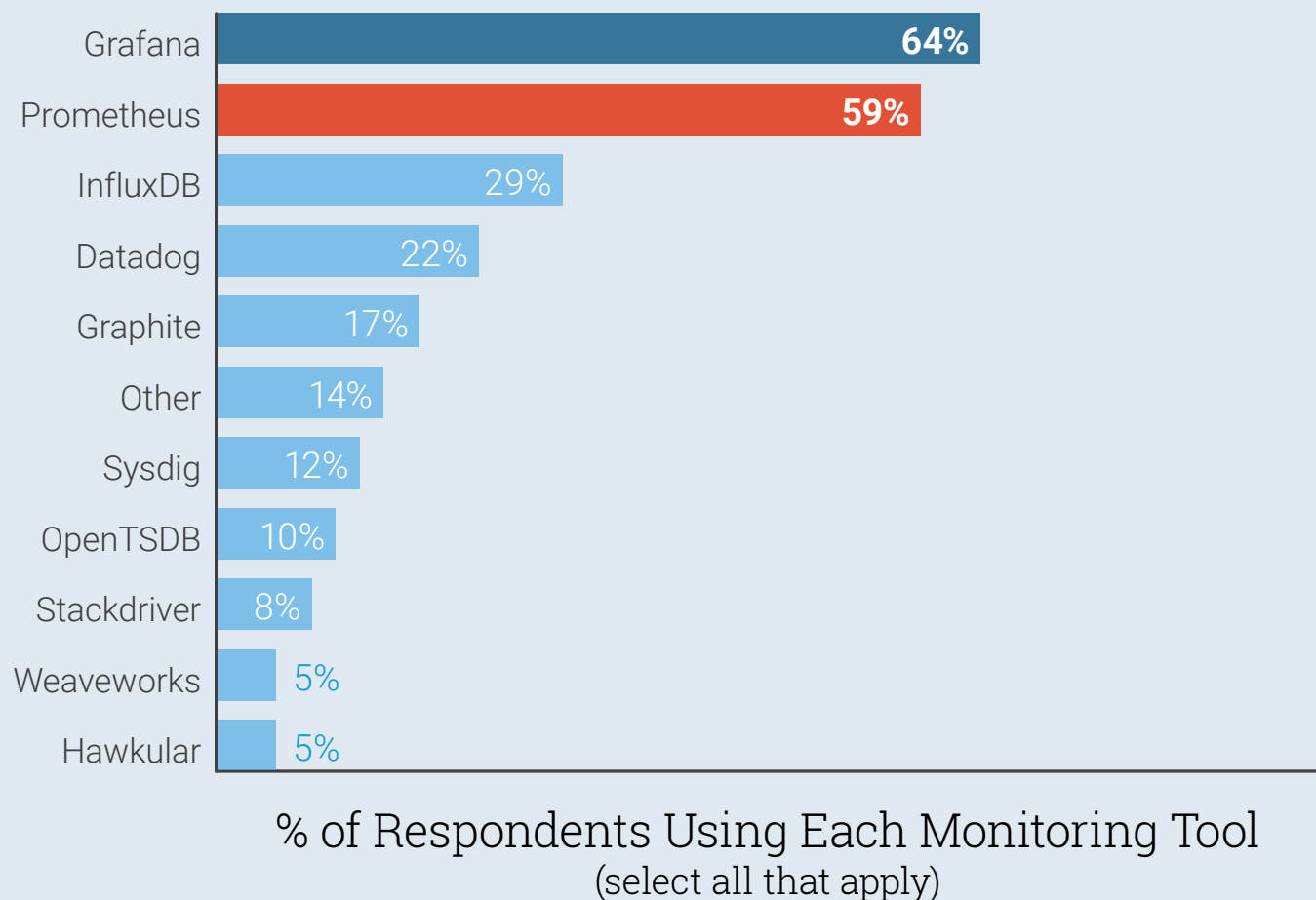
Prometheus also has some drawbacks: Accuracy is one of them. Prometheus scrapes data and such scrapes are not guaranteed to occur. If you have services that require accuracy, such as per-usage billing, then Prometheus is not a good fit. It also doesn't work well for non-HTTP systems. HTTP is the dominant 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](#). 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 New Relic. (See the second ebook in this series, [Kubernetes Deployment & Security Patterns](#), for a more detailed analysis.)

Grafana and Prometheus Are the Most Widely Used Tools for Monitoring Among Kubernetes Users



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.

© 2018 THE NEW STACK

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

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.

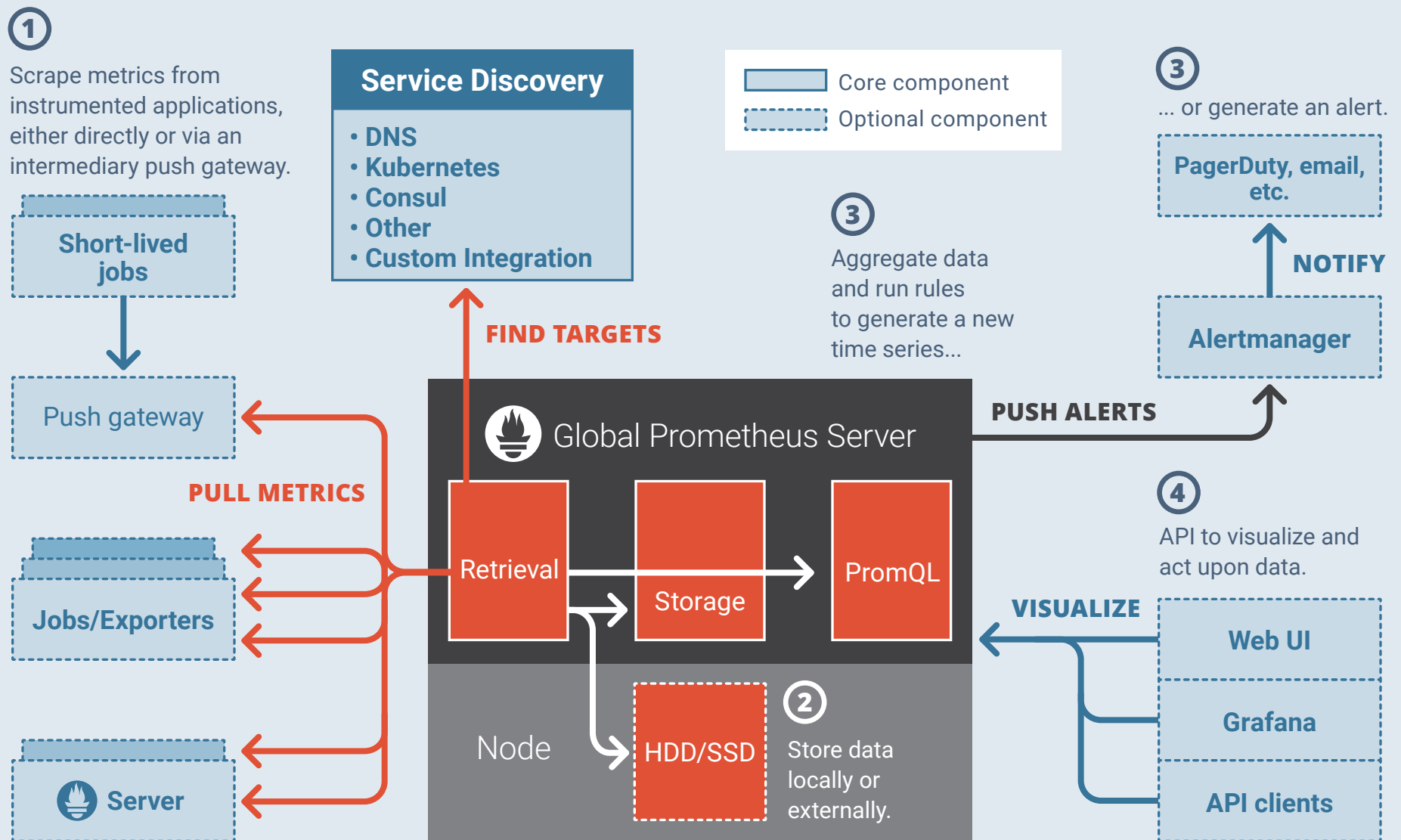
Components and Architecture Overview

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

Prometheus Ecosystem Components



Source: <https://prometheus.io/docs/introduction/overview/>

© 2018 THE NEW STACK

FIG 4.6: Components outside of the Prometheus core provide complementary features to scrape, aggregate and visualize data, 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 work-around 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 a 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:

1. Storing, aggregating and de-duplicating alerts.
2. Inhibiting and silencing alerts.
3. 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 precompute 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](https://hub.docker.com/prom/prometheus). A step-by-step guide to install Prometheus is available on the Prometheus [website](https://prometheus.io/docs/operating/).

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.

CLOSING

The narrative about continuous integration and continuous delivery in Kubernetes starts with DevOps. It encompasses the new drive for faster and continuous deployment, and a deeper understanding for how to manage components running on microservices. The transition to modern, application-oriented architectures inevitably leads organizations to find people with the DevOps experience needed to manage Kubernetes and relevant cloud-native services.

As teams grow, we now see more of this need for declarative infrastructure. Application architectures built on DevOps practices work better and run with less friction, but in the end, they just help make the infrastructure boring. If it is boring, then great — it's working. Then the developer has more control over their own resources, and the performance of the application becomes the primary focus. The better the performance, the happier the end user and the more uniform the feedback loop between users and developers. In this way, cloud-native technologies, such as Kubernetes, provide game-changing business value.

With great execution can come great results. But the scope has changed. There are historical barriers to overcome that inhibit Kubernetes use, namely the social issues that surface when people from different backgrounds and company experiences enter an open source project and work together. The Kubernetes community is maturing, and defining values has become a priority as they work to strengthen the project's core. Still, the downside to Kubernetes does have to be taken into context when thinking through longer term business and technical goals. It is imperative to have a trust in the Kubernetes project as it matures. There will be conflicts and stubbornness. And it will all be deep in the project, affecting testing and the ultimate delivery of updates to the Kubernetes engine. It's up to the open source communities to work through how the

committees and the Special Interest Groups align to move the project forward. It's a problem that won't go away. Here, too, the feedback loop becomes critical between users and the Kubernetes community.

In this comes some wisdom to glean about the nature of continuous delivery and how it may change with the evolution of CI/CD platforms, and new uses for Git to manage Kubernetes operations. These efforts encompass discussions about security, identity, service meshes and serverless approaches to use the resources in further abstracted manners.

Only when the abstraction becomes dysfunctional does true change come. There has to be a continual feedback loop throughout the build and deploy cycle to better know how the comparison of time-series information shows anomalies, for example. Emerging patterns become the best way to find clues to problems. Feedback loops are also key for the developer experience, which is critical in order for them to build their own images in the best manner possible. What's become obvious to us in editing this ebook, is that this feedback loop must be present both between users and the Kubernetes community itself, and within the organizations that build and deploy their applications on top of it.

Coming up next for The New Stack is a new approach to the way we develop ebooks. Look for books on microservices and serverless this year with corresponding podcasts, in-depth posts, and activities around the world wherever pancakes are being served.

Thanks and see you again soon.

Alex Williams

Founder, Editor-in-Chief

The New Stack

DISCLOSURE

In addition to our ebook sponsors, the following companies are sponsors of The New Stack:

Alcide, AppDynamics, Blue Medora, Buoyant, CA Technologies, Chef, CircleCI, Cloud Foundry Foundation, {code}, InfluxData, Mesosphere, Microsoft, Navops, New Relic, OpenStack Foundation, PagerDuty, Pivotal, Portworx, Puppet, Raygun, Red Hat, Rollbar, SaltStack, StackRox, The Linux Foundation, Tigera, Twistlock, Univa, VMware, Wercker and WSO2.

