vmware PRESS

Accelerating Digital Transformation with Containers and Kubernetes

An Introduction to Cloud-Native Technology

Author
Steve Hoenisch

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors, VMware Press, VMware, and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

The opinions expressed in this book belong to the author and are not necessarily those of VMware.

VMware, Inc.
3401 Hillview Avenue
Palo Alto CA 94304
USA Tel 877-486-9273
Fax 650-427-5001
www.vmware.com.

Contents

Introduction 8
- Organization of this Book 8
- Point of Departure: Cloud-Native Terminology 9

Driving Digital Transformation with Containers and Kubernetes 13
- The Business Value of Digital Transformation 13
- Cloud-Native Applications 13
- 12-Factor Apps: A Methodology for Delivering Software as a Service 15
- The Business Value of Kubernetes 17
- An Example Use Case 17

Demystifying Kubernetes 19
- Platform vs. Runtime Environment 19
- Robust Open-Source Technology from a Google Production System 19
- Defogging the Abstract Terminology of Kubernetes 20
- A Concise Overview of Kubernetes 21
- Just Another Fad in the Hype Cycle? 24
- Kubernetes in Production Environments 24
- A Rapidly Maturing Ecosystem 25
- Kubernetes Won't Solve All Your Problems 25

Introduction to Cloud-Native Architectures and Practices 27
- Microservices 27
- Deconstructing the Monolith and Other Use Cases 29
- Kubernetes for Cloud-Native and 12-Factor Applications 30
- Profile of a DevOps Engineer: Responsibilities and Skills 32
- Continuous Integration and Continuous Deployment 34

Container Technology in the Software-Defined Data Center 35
- VMware vSphere and the SDDC 36
- Abstract and Automate: Network Virtualization 36
- Risk-Free Scale Out with Ease: Virtual Storage 37
- Put a Lid on It: Security for Containers 38
- Linux Container Hosts 41
- Securing Cloud Platforms with Lightwave 44
- Managing Container Images with Harbor 51
- Microservices Meets Micro- segmentation: Delivering Developer-Ready Infrastructure for Modern Application Development with NSX 59

BOSH 60
CFCN for Deploying and Operating Kubernetes 67

Container Platforms and Services 70
High-Level Use Cases for Container Platforms. 70
Maturity of Container Adoption. 70
Cloud Natives 71
Matching the Platform to the Project 71
Prescription and Complexity 71
vSphere Integrated Containers 73
VMware Pivotal Container Service 92

Use Cases 101
Self-Service Infrastructure for Agile Development 101
Replatforming Applications with PKS 106
Deploying New Cloud-Native Apps with PKS 108
Exploiting the Power of Containers 109
Running a Containerized App with
Photon OS on Amazon Elastic Cloud Compute. 109
Using vSphere Integrated Containers to Solve
Container Networking Problems 120
Providing Persistent Storage for Legacy Applications. 124
Setting Up a Developer Sandbox 133
Deploying Jenkins by Using VIC 141
Optimizing Cloud-Native Apps with PCF and
Developer-Ready Infrastructure from VMware. 147
Case Study: Optimizing Critical Banking Workloads 158

Conclusion 162

Glossary. 163

Author and Contributors

Author and Editor

Steve Hoenisch is a technology evangelist, writer, and editor who specializes in emerging technology and cloud-native solutions. He's written numerous influential technical white papers and magazine articles on digital transformation, Kubernetes, containers, big data, Hadoop, storage platforms, security, and regulatory compliance. A former newspaper editor with a master's degree in linguistics, he has published articles in XML Journal, The Hartford Courant, and the Chicago Tribune. He works in the Cloud-Native Apps business unit at VMware.

Contributors

Ben Corrie has been a leading voice of technical innovation in the container space at VMware for three years. Ben was the initiator of the research that led to the vSphere Integrated Containers product and as an architect on that product, Ben's role now is to look 6 to 12 months ahead to help align VMware with the container challenges ahead.

Patrick Daigle is a Senior Technical Marketing Architect in Montreal, Canada. As part of VMware's Cloud-Native Applications business unit, he focuses on vSphere Integrated Containers and works with enterprises around the globe, explaining and demonstrating the benefits of VMware container solutions and how they can bring value to the business.

Contributors, cont.

Ning Ge is a Senior Product Marketing Manager in VMware's Cloud-Native Apps business unit and works on VMware's container solutions, such as VMware Pivotal Container Service and vSphere Integrated Containers. Ning has over 7 years of experience in marketing enterprise software technologies and solutions, and her main area of focus includes container and cloud-native, middleware, and infrastructure technologies. Ning has master's degrees in both Business Administration and Communications.

Merlin Glynn is a product manager at VMware, where he builds products that help customers architect and deploy cloud-native applications. Merlin has been building complex environments as an architect for over 20 years, focusing on solutions for large enterprises and the academic and scientific community. Previously, he architected some of the world's largest supercomputers at IBM, which were regularly listed on the TOP500 list, and, at Pivotal, designed many next-generation Pivotal Cloud Foundry platforms for key enterprise customers. Merlin is a certified AWS Solutions Architect. He enjoys volunteering for charities.

Simone Morellato is currently a Director of Technical Product Management at VMware where he leads technical product management and marketing efforts for the company's Cloud-Native Applications business unit. Simone has more than 16 years of experience in storage, networking and infrastructure for both traditional and cloud applications. Before joining VMware, Simone worked at Apcera, a container management platform, acquired by Ericsson. He has also held leadership, marketing and technical presales roles at Cisco, Riverbed Technology, Astute Networks and Andiamo Systems (later acquired by Cisco).

Tom Scanlan is a Senior Consultant in the CNA/DevOps and Emerging technologies arenas. He has 20 years experience across software engineering, systems and networking administration and consulting thereon. Tom has had deep focus on enabling DevOps and multicloud architectures.

Introduction

Digital transformation, the commoditization of IT, the Internet of things, the proliferation of mobile devices, the growing popularity of public clouds, big data, and other seismic technological changes are radically altering the way businesses are run. Innovative software applications are, for many businesses, a critical objective. Consumers, customers, and keeping ahead of the competition demand it.

But one-time innovation is often not enough. The digital era calls for continuous innovation at an accelerated pace—and the kind of modernized data centers and software development technologies that make it possible.

Container technology can help transform a company into a digital enterprise focused on delivering innovations at the speed of business. Containers package applications and their dependencies into a distributable image that can run almost anywhere, streamlining the development and deployment of software.

By adopting containers, organizations can take a vital step toward remaking themselves into flexible, agile digital enterprises capable of accelerating the delivery of innovative products, services, and customer experiences. Enterprises can become the disrupters instead of the disrupted.

But containers create technology management problems of their own, especially when containerized applications need to be deployed and managed at scale, and that's when Kubernetes comes into play. Kubernetes automates the deployment and management of containerized applications. More specifically, Kubernetes orchestrates containerized applications to manage and automate resource utilization, failure handling, availability, configuration, desired state, and scalability.

This book introduces you to containers and Kubernetes, explains their business value, explores their use cases, and illuminates how they can accelerate your organization's digital transformation.

Organization of this Book

The chapters at the beginning of the book explain the business value container technology and examine how enterprises are modernizing their data centers to take advantage of cloud-native innovations.

After briefly examining the architectural patterns, practices, processes, and pipelines that help propel you toward digital transformation, the book

considers the kind of infrastructure, virtualization technologies, systems, and security required by next-generation data centers.

The chapters that follow become increasingly technical as they use two key products from VMware—VMware vSphere Integrated Containers and VMware Pivotal Container Service—to explain the architecture of cloud-native applications, the capabilities of Kubernetes, and the use cases for container technology.

The final sections of the book turn to examples that demonstrate how to exploit the power of containers and Kubernetes to solve technical problems.

Point of Departure: Cloud-Native Terminology

Container technology comes with its own lexicon. If you're familiar with the basic terminology around containers, Kubernetes, and cloud-native applications, you can skip this section. For plain-language descriptions of terminology in the cloud-native space, see the glossary at the end of the book.

Containers

Container: A portable format, known as an image, for packaging an application with instructions on how to run it as well as an environment in which the image is executed. When the container image is executed, it runs as a process on a computer or virtual machine with its own isolated, self-described application, file system, and networking. A container is more formally known as an *application container*. The use of containers is increasing because they provide a portable, flexible, and predictable way of packaging, distributing, modifying, testing, and running applications. Containers speed up software development and deployment.

Docker is a widely used container format. Docker defines a standard format for packaging and porting software, much like ISO containers define a standard for shipping freight. As a runtime instance of a Docker image, a container consists of three parts:

- A Docker image
- An environment in which the image is executed
- A set of instructions for running the image

Containerized application: An application that has been packaged to run in a container.

Kubernetes and Orchestration

Kubernetes: A system that automates the deployment and management of containerized applications. As an application and its services run in containers on a distributed cluster of virtual or physical machines, Kubernetes manages all the moving pieces to optimize the use of computing resources, to maintain the desired state, and to scale on demand. On Kubernetes, a container (or a set of related containers) is deployed in a logical unit called a pod. In addition to scheduling the deployment and automating the management of containerized applications, a key benefit of Kubernetes is that it maintains the desired state—the state that an administrator specifies the application should be in.

Cluster: Three or more interconnected virtual machines or physical computers that, in effect, form a single system. A computer in a cluster is referred to as a node. An application running on a cluster is typically a distributed application because it runs on multiple nodes. By inherently providing high availability, fault tolerance, and scalability, clusters are a key part of cloud computing.

Orchestration: Because it can automatically deploy, manage, and scale a containerized application, Kubernetes is often referred to as an orchestration framework or an orchestration engine. It orchestrates resource utilization, failure handling, availability, configuration,desired state, and scalability.

Application Types and Architectural Patterns

Microservices: A "modern" architectural pattern for building an application. A microservices architecture breaks up the functions of an application into a set of small, discrete, decentralized, goal-oriented processes, each of which can be independently developed, tested, deployed, replaced, and scaled.

Cloud-native applications: Generally speaking, they are developed and optimized to run in a cloud as distributed applications. According to the Cloud Native Computing Foundation, cloud-native applications, which are also generally referred to as "modern" applications, are marked by the following characteristics:

- Containerized for reproducibility, transparency, and resource isolation.

- Orchestrated to optimize resource utilization.
- Segmented into microservices to ease modification, maintenance, and scalability.

Cloud-native applications are typically developed and deployed on a containers as a service platform (CaaS) or a platform as a service (PaaS).

12-factor app: A methodology for developing a software-as-a-service (SaaS) application—that is, a web app—and typically deploying it on a platform as a service or a containers as a service.

Platforms

The overarching business objective of using a container platform is to accelerate the development and deployment of scalable, enterprise-grade software that is easy to modify, extend, operate, and maintain. Three types of platforms provide varying degrees of support for container technology:

- A platform for running individual container instances.
- Containers as a service.
- Platform as a service.

A *platform as a service* is often referred to simply as an application platform. In this context, an application platform helps developers not only write code but also integrate tools and services, such as a database, with their application as microservices. An example of a private platform as a service that is also referred to as an application platform is Pivotal Cloud Foundry.

A *container-as-a service* platform helps developers build, deploy, and manage containerized applications, typically by using Kubernetes or another orchestration framework. An example of a container as a service platform is VMware Pivotal Container Service.

A platform for running container instances helps developers build and test a containerized application. It does not, however, orchestrate the containerized application with Kubernetes, nor does it provide a service broker so that developers can integrate tools, databases, and services with an app. An example of a container instance platform is VMwarevSphere Integrated Containers.

Platforms and Developer Operations

Delivering software in an expedient, reliable, sustainable way requires collaboration between IT teams and developers. *DevOps* takes place when developers and IT come together to focus on operations in the name of streamlining and automating development and deployment. DevOps is a key practice driving cloud-native applications.

To help DevOps, a container platform provides some or all of the following services:

- Lets developers add tools and services to their app through a service broker or catalog.
- Adds security, logging, monitoring, analytics, dashboards, maintenance, and other operational features.
- Provides container networking.
- Exposes an API.
- Automates some or all delivery and deployment processes.
- Eases continuous integration, continuous delivery, and continuous deployment.

Continuous Integration, Delivery, and Deployment

Continuous integration constantly combines source code from different developers or teams into an app and then tests it. *Continuous delivery* readies an application or part of an application for production by packaging and validating it. *Continuous deployment* automatically deploys an application or part of an application into production. The entire process forms the CI/CD pipeline when the *D* in the abbreviation is assumed to represent deployment.

Digital Transformation

All the modernizing elements covered in this section—containers, Kubernetes, microservices, container platforms, DevOps, and the CI/CD pipeline—converge into a powerful recipe for digital transformation: You can optimize the use of your computing resources and your software development practices to extend your enterprise's adaptability, productivity, innovation, competitive advantage, and global reach.

THE BUSINESS BENEFITS OF MODERNIZATION

The use of containers, microservices, and Kubernetes modernizes application development, yielding business benefits that ultimately bolster your competitive advantage:

- Shorten software's time to market
- Improve developer agility and productivity
- Respond faster to change

Driving Digital Transformation with Containers and Kubernetes

Container technology is a key contributing factor to achieving digital transformation. This chapter connects the dots between containers and Kubernetes on the one hand and digital transformation and business value on the other.

The Business Value of Digital Transformation

The reasons enterprises are undergoing digital transformation are clear:

- Create new applications that engage customers in innovative and captivating ways.
- Improve operations to more efficiently deliver better products and services at a lower cost to the business.
- Generate new revenue streams by rapidly adapting to changes in market conditions and consumer preferences.

The ingredients for building effective applications are less clear than the desired outcomes.

To be effective in this era, applications require an architecture that fosters fluid, rapid, responsive development and deployment while still maintaining the security, performance, and cost-effectiveness of established patterns. Containers provide the basis for a new application architecture that supports digital transformation and lays the foundation for innovation. Organizations that are adopting containers see them as a fast track to building and deploying cloud-native applications and twelve-factor apps.

Cloud-Native Applications

The Cloud Native Computing Foundation, a project of The Linux Foundation, defines cloud-native applications as follows:[1]

1. Containerized—Each part (applications, processes, etc.) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.

[1] This definition is from the FAQ of the Cloud Native Computing Foundation, https://www.cncf.io/about/faq/.

2. Dynamically orchestrated—Containers are actively scheduled and managed to optimize resource utilization.
3. Microservices oriented—Applications are segmented into microservices. This segmentation significantly increases the overall agility and maintainability of applications.

Kubernetes covers the second part of the definition by scheduling and managing containers. For the third part, both Kubernetes and Docker help implement microservices.

The key element, however, is the container—a process that runs on a computer or virtual machine with its own isolated, self-described application, file system, and networking. A container packages an application in a reproducible way: It can be distributed and reused with minimal effort.

DOCKER CONTAINER DEFINED

With containers, Docker has defined a standard format for packaging and porting software, much like ISO containers define a standard for shipping freight. As a runtime instance of a Docker image, a container consists of three parts:

- A Docker image
- An environment in which the image is executed
- A set of instructions for running the image

—Adapted from the Docker Glossary

Docker containers are the most widely deployed container. A manifest, called a Dockerfile, describes how the image and its parts are to run in a container on a host. To make the relationship between the Dockerfile and the image concrete, here's an example of a Dockerfile that installs MongoDB on an Ubuntu machine running in a container. The lines starting with a number sign are comments describing the subsequent commands.

```
# MongoDB Dockerfile from https://github.com/dockerfile/mon-
godb
# Pull base image.
FROM dockerfile/ubuntu
# Install MongoDB.
RUN \
  apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv 7F0CEB10 && \
  echo 'deb http://downloads-distro.mongodb.org/repo/ubun-
tu-upstart dist 10gen' > /etc/apt/sources.list.d/mongodb.
list && \
```

```
  apt-get update && \
  apt-get install -y mongodb-org && \
  rm -rf /var/lib/apt/lists/*
# Define mountable directories.
VOLUME ["/data/db"]
# Define working directory.
WORKDIR /data
# Define default command.
CMD ["mongod"]
# Expose port 27017 for the process and port 28017 for http
EXPOSE 27017
EXPOSE 28017
```

12-Factor Apps: A Methodology for Delivering Software as a Service

In contrast to the cloud-native application, the 12-factor app is defined as much by its processes as by its systemic properties. It is a methodology for developing a software-as-a-service (SaaS) application—that is, a web app—and typically deploying it on a platform-as-a-service (PaaS), such as Pivotal Cloud Foundry. Here are the 12 factors with a brief explanation of each one:[2]

1. Deploy the application many times from one codebase. The codebase is stored in a repository, managed with a version control system such as Git as it is modified, and then deployed many times as a running instance of the app from the same codebase. As a result, a deployment is often running in three environments: on each developer's local environment, in a staging environment, and in the production environment.

2. Declare and isolate dependencies. The app does not implicitly rely on system-wide packages; instead, it declares the dependencies in a declaration manifest. Explicitly declaring dependencies makes it easier for new developers to set up their development environment.

3. Store the configuration in the environment, not the code. For configuration information that varies by deployment, the app stores the information in environmental variables. The environmental variables are granular controls that are managed independently for each deployment so that the app can easily scale into more deployments over time.

4. Connect to supporting services, such as a database or a storage system, instead of including it in the code. The app treats such

[2] The twelve factors are paraphrased from the descriptions at the Twelve-Factor App web site.

services as resources that can be attached to or detached from a deployment by modifying the configuration.

5. Treat build and run as separate stages. A deployment of the codebase takes place in three separate stages: build, release, and runtime. The build stage converts the codebase into an executable—a build—and then the release stage combines the build with the configuration to produce a release that's ready for execution in the runtime environment.
6. Run the app as stateless processes. The processes share nothing with other processes, and data that must persist is stored in a database running as a stateful supporting service.
7. Expose services by using port binding. Taking HTTP as an example, the app exports HTTP as a service by binding to a port and listening on the port for incoming requests.
8. Scale out by adding concurrent processes. The app handles workloads by assigning each type of work to a process type. A web process, for example, handles HTTP requests, while a worker process manages background tasks.
9. Ensure durability with disposability. Processes are disposable—they can be started or stopped quickly to make sure that the application can be changed or scaled easily.
10. Make development and production peers. The app is geared toward continuous deployment by allowing developers to integrate new code quickly and to deploy the app themselves in a production environment. The production and development environments should be as similar as possible.
11. Process logs as event streams. The app neither routes nor stores the output stream from its logs but instead writes it as a stream of data to standard output, where it is to be collected by the execution environment and routed to a tool or system, such as Hadoop, for storage or analysis.
12. Run one-off management scripts and tasks, such as a database migration, in an environment identical to that of the app's long-running processes.

Containers and Kubernetes help satisfy aspects of these imperatives. Containers, for example, play a key role in 12-factor apps by letting you declare and isolate dependencies. Containers also help ensure durability with disposability by, among other things, starting quickly and stopping gracefully. Many of the other factors are supported by Kubernetes.

The Business Value of Kubernetes

Kubernetes uses its architecture and capabilities to manage containerized applications in a distributed cluster. The results help fulfill the business promise of digital transformation:

- Kubernetes makes it easier and cheaper to run applications in public, private, or hybrid clouds.
- Kubernetes accelerates application development and deployment.
- Kubernetes increases agility, flexibility, and the ability to adapt to change.

ADVANTAGES OF USING KUBERNETES

- Consolidate servers and reduce costs through efficient resource utilization.
- Ease and expedite application deployment.
- Decouple applications from machines for portability and flexibility.
- Easily modify, update, extend, or redeploy applications without affecting other workloads.
- Elegantly handle system faults and machine failures through automation, self-healing. and high availability.
- Automate scalability for containerized applications.

An Example Use Case

A short case study provides a high-level use case for managing containers with Kubernetes.

A taxicab company in a major metropolitan area is losing riders to car-sharing services, imperiling its once-strong local market share. It needs to transform itself into a digital enterprise capable of competing with car-sharing companies. To do so, the company wants to develop its own mobile app, cost-effectively run the app in its modest data center, and attempt to provide innovative services.

To its credit, the taxi company retains a number of advantages: a well-known, long-established local brand with a reputation for timely, courteous, safe drivers.

As recently hired developers work on the mobile app, the taxi company modernizes its data center with commodity hardware and virtualization. To maximize resource utilization of its small data center and to minimize

costs, the company plans to run its new app in Docker containers on virtual machines. Kubernetes will orchestrate the containerized application.

After being rolled out and advertised in and on its cars, the app is an instant success. To meet fluctuations in use of the app, the company uses Kubernetes to dynamically scale the number of containers running the app. For example, when metrics for the app hit a predefined threshold indicating high usage, which typically happens during rush hour, the company's DevOps team uses the horizontal pod autoscaling feature of Kubernetes to automatically maximize the number of containers so that the system can match demand. At 4 am, in contrast, the number of containers is reduced to elastically match the low demand at that time, conserving resources.

The mobile app correlates ride requests with location. By mining the data and combining it with its intimate historic knowledge of the city's patterns, the cab company can station cabs in the perfect locations for hailing customers—preempting some car requests to the competition. What's more, because the company processes the app's logs as event streams, the company can do this dynamically during day and night, shifting cars to hot spots.

Because the company implemented the app by using containers, developers can roll out new changes daily. The data that the app collects helps the company pinpoint new features and quickly innovate to focus on its strengths, such as identifying recurring customers and rolling out a rewards program to retain them.

The business benefits of the company's technical agility, containerized application, and Kubernetes orchestration add up to a competitive advantage:

- The scheduling policies in Kubernetes give the company the elasticity it needs to dynamically match demand in a cost-effective way with its modest but now-modernized data center.
- Faults and failures are handled automatically by Kubernetes, reducing troubleshooting demands on its small DevOps staff.
- The seamless modification of the app and its features helps the company beat its bigger, less local rivals by being more agile and better able to apply its knowledge of local patterns.
- Containers and Kubernetes make it easier and cheaper to run the app.
- The ease with which the DevOps team can port containers from the test environment to production accelerates the development and deployment of new features.

Demystifying Kubernetes

As an emerging technology with a Greek name, Kubernetes carries mythical connotations. Some of its features only add to the suspicion of magic—the meaning of capabilities like automatic binpacking, horizontal scaling, self-healing, and secret management might not be readily apparent. The sense of power that these terms engender, however, seems palpable: The potential to automatically place, pilot, scale, and heal an application in secret would turn the head of anyone working in IT.

This chapter aims to demystify Kubernetes by presenting a concise overview of the platform and by addressing some of the common misconceptions surrounding the platform. Here you'll find brief explanations of what it is, what it isn't, how it works, what it does, and why you should care.

Platform vs. Runtime Environment

Kubernetes is not a runtime environment. It is a platform for managing, or orchestrating, application containers. The platform deploys, scales, and operates containers.

As an application and its services run in containers on a distributed cluster of virtual or physical machines, Kubernetes choreographs all the moving pieces so they operate in a synchronized way to optimize the use of computing resources and to maintain the correct state.

Maintaining the desired state of a distributed application running in containers is one of the key value propositions of Kubernetes—you specify the state you want the application to be in, and Kubernetes manages all the application's services and resources to establish and maintain that desired state.

In Kubernetes, the container runtime itself is typically provided by Docker, but you can optionally use other container runtimes, such as rkt (pronounced the same as the word rocket). In other words, containers have their own runtime.

Although you don't need Kubernetes to use containers, you will likely need Kubernetes if you want to robustly and repeatedly deploy and automate a containerized application in a production environment.

Robust Open-Source Technology from a Google Production System

Kubernetes started out as a closed-source project at Google based on an orchestration system called Borg. Google uses Borg to initiate, schedule, restart, and monitor public-facing applications, such as Gmail and Google Docs, as well as internal frameworks, such as MapReduce.[3] Kubernetes was heavily influenced by Borg and the lessons learned from running Borg on a massive scale in a production environment. In 2015, Google open-sourced Kubernetes. Shortly afterward, Google donated it as seed technology to the Cloud Native Computing Foundation, a newly formed open-source project hosted by the Linux Foundation. (VMware is a member of the Linux Foundation and the Cloud Native Computing Foundation.)

A burgeoning open-source ecosystem around Kubernetes is rapidly evolving. A project called Prometheus adds monitoring; containerd and rkt provide alternative container runtimes; linkerd establishes a service mesh; and a number of other projects cover additional requirements, such as logging and service discovery. A open source project called Cloud Foundry Container Runtime, formerly known as Kubo, brings the industrial-strength release engineering, deployment, and lifecycle management capabilities of BOSH to Kubernetes.

Defogging the Abstract Terminology of Kubernetes

Terminology is partly responsible for enshrouding Kubernetes in myth. Even the name itself sounds somewhat mythical—it's the Greek word for helmsman or pilot. But there's an assortment of other terms that help push the system's intelligibility into the shadows: pod, kubelet, replica set, NodePort, horizontal autoscaler, and stateful set.

Other terms, abbreviations, and acronyms taint the fringes of the Kubernetes platform as it bumps up against containers on the one hand and the accompanying infrastructure on the other: runC, OCI, YAML, JSON, IaaS, PaaS, and KaaS. There's even the odd abbreviation of Kubernetes itself: K8s.

Yet once you become familiar with the system, its relationship to containers, and the infrastructure at its edges, the meaning of the terms comes into focus.

---

[3]For more on Borg, see Research at Google, *Large-Scale Cluster Management at Google with Borg,* 2015.

On Kubernetes, a pod is the smallest deployable unit in which one or more containers can be managed—in other words, you run a container image in a pod. A set of pods typically wraps a container, its storage resources, IP address, and other options up into an instance of an application that will run on Kubernetes. Docker is usually the container runtime used in a pod. As a Kubernetes administrator, you specify a pod by using a YAML file.

Another fundamental term in Kubernetes is kubelet. It manages pods. The lifecycle of pods is in turn managed by a replica set. And when a pod provides a service, such as a web server, a NodePort presents the service on a port on the nodes in the cluster for external access. When requests of that service exceed a threshold, the horizontal pod autoscaler adds resources to handle the increase in demand. If the service happens to be a stateful application running in a set of pods, the stateful set allocates and manages resources for the stateful pods, such as persistent storage.

Some terms repeatedly come up in relation to containers or infrastructure. runC refers to the code module that launches containers; it is part of containerd and managed by OCI, which stands for Open Container Initiative, an organization dedicated to setting industry-wide container standards. IaaS stands for infrastructure as a service; PaaS stands for platform as a service; and KaaS stands for Kubernetes as a service.

An example of a platform as a service is Pivotal Cloud Foundry, which in turn requires elastic infrastructure as a service—such as VMware vSphere® or a VMware software-defined data center—to meet its resource demands.

For definitions of more Kubernetes terms, see the glossary at the end of the book.

A Concise Overview of Kubernetes

Google originally developed Kubernetes. The company uses its predecessor, called Borg, to initiate, schedule, restart, and monitor public-facing applications, such as Gmail and Google Docs, as well as internal frameworks, such as MapReduce.[4] Based on Google's original system plus enhancements from the lessons learned with Borg, Kubernetes can work in your data center, across clouds, and in a hybrid data center. Kubernetes automatically places workloads, restarts applications, and adds resources to meet demand.

---

[4] *Large-Scale Cluster Management at Google with Borg,* Research at Google, 2015.

Here, briefly, is how it works. A Kubernetes cluster contains a master node and several worker nodes. Then, when you deploy an application on the cluster, the components of the application run on the worker nodes. The master node manages the deployment.

Main Components

Kubernetes includes these components:

- The Kubernetes API
- The Kubernetes command-line interface, kubectl
- The Kubernetes control plane

The control plane comprises the processes running on the Kubernetes master and on each worker node. On the master, for example, Kubernetes runs several processes: the API server, the controller, the scheduler, and etcd. The worker nodes run the kubelet process to communicate with the master and the proxy process to manage networking.

Kubernetes Object Model

One of the keys to the Kubernetes system is how it represents the state of the containerized applications and workloads that have been deployed. Kubernetes represents state by using "objects," such as service, namespace, and volume. These objects are typically set by an object specification, or spec, that you create for your cluster.

In the Kubernetes object model, the concept of a Pod is the most basic deployable building block. A Pod represents an instance of an app running as a process on a Kubernetes cluster. Here's where the Docker runtime comes back into the equation—Docker is commonly used as the runtime in a Kubernetes Pod.

Kubernetes also includes Controllers that implement most of the logic in Kubernetes. The Controllers provide features such as the replica set and the stateful set.

Maintaining the Desired State

The Kubernetes control plane manages the state of all these objects to ensure that they match your desired state. You can specify a desired state by creating an object specification for a service with a YAML file. Here's an example:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-demo-service
  labels:
    app: nginx-demo
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: nginx-demo
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-demo
spec:
  replicas: 3
  template:
    metadata:
   labels:
    app: nginx-demo
   spec:
    containers:
    - name: nginx-demo
      image: myrepo/nginx
    ports:
        - containerPort: 80
```

When you submit this file to the Kubernetes master with the kubectl command-line interface, the Kubernetes control plane implements the instructions in the file by starting and scheduling applications so that the cluster's state matches your desired state. The Kubernetes master and the control plane then maintain the desired state by orchestrating the cluster's nodes, which can be actual servers or virtual machines.

The core of the architecture is an API server that manages the state of the system's objects. The API server works with Kubernetes subcomponents, or clients, that are built as composable microservices, such as the replication controller specified in the YAML file. The replication controller regulates the desired state of pod replicas when failures occur.

MANAGING CONTAINERIZED APPLICATIONS WITH KUBERNETES

- Kubernetes orchestrates distributed, containerized applications to:
- Optimize utilization of computing resources.
- Provide policies for scheduling.
- Maintain desired state.
- Handle faults and failures with automation.
- Provide high availability.
- Monitor jobs in real-time.
- Manage an application's configuration.
- Dynamically scale to meet changes in demand.

Just Another Fad in the Hype Cycle?

In July, Kubernetes celebrated its second anniversary. Kubernetes is among the highest velocity cloud-related open-source development projects in the world; for a listing of facts and figures detailing the project's popularity and adoption, see the Kubernetes retrospective.

In addition, the membership of the Cloud Native Computing Foundation, which is the open-source group managing Kubernetes, has attracted major players in the cloud-computing space, including Dell Technologies, IBM, Amazon, Microsoft, Google, Intel, AT&T, and many more. End-user members include Twitter, Capital One, eBay, and Goldman Sachs. For a list of members, see the Cloud Native Computing Foundation web site.

Kubernetes in Production Environments

In fact, numerous organizations have deployed Kubernetes in production environments. Although it is an emerging technology with a burgeoning ecosystem, the feature set and API of Kubernetes are robust for a two-year-old open-source project. Keep in mind, though, that the predecessor of and the principles behind Kubernetes have been running in production at Google since 2005, orchestrating applications such as Gmail in Google's cloud-scale production environment. The Kubernetes website contains several case studies that detail how different organizations have adopted Kubernetes.

In addition, surveys in both 2016 and 2017 showed not only large growth in the adoption of containers but also significant increases in the number of organizations using Kubernetes.[5]

Implementing Kubernetes in production, however, is likely to require the addition of other projects and tools in the container ecosystem.

A Rapidly Maturing Ecosystem

Fortunately, the container ecosystem is rapidly maturing. A clear marker of that increasing maturity is the expansion of projects hosted by the Cloud Native Computing Foundation. To support Kubernetes deployments, the foundation hosts key open-source projects, including the following:

- Prometheus, a monitoring system for Kubernetes
- OpenTracing, a vendor-neutral standard for distributed tracing
- Fluentd, a data collector for unified logging
- linkerd, a service mesh that adds service discovery, routing, failure handling, and visibility to cloud-native applications

In addition, there are a variety of enterprise-grade, production-ready technologies for working with Kubernetes in a software-defined data center, such as VMware vRealize® Log Insight™, which can process a container's standard output as a data stream.

Kubernetes Won't Solve All Your Problems

Kubernetes probably won't solve all your IT, application development, and deployment problems. But as your organization undergoes digital transformation, Kubernetes might solve some of the most pressing challenges in deploying and managing applications at scale.

Production deployments of Kubernetes show that it delivers substantial IT business benefits as well as benefits to a business's bottom line. In summary, here are some of the benefits for IT, system administrators, and DevOps:

---

[5]: See Portworx, *"2017 Annual Container Adoption Survey: Huge Growth in Containers,"* April 12, 2017; April 12, 2017; ClusterHQ, *"Container Market Adoption Survey 2016";* Sysdig, *"The 2017 Docker Usage Report,"* Apurva Dave, April 12, 2017; and Forbes, *"2017 State of Cloud Adoption and Security,"* Louis Columbus, April 23, 2017.

- Consolidate servers and reduce costs through efficient resource utilization.
- Elegantly handle machine failure through self-healing and high availability.
- Ease and expedite application deployment, logging, and monitoring.
- Automate scalability for containers and containerized applications.
- Decouple applications from machines for portability and flexibility.
- Easily modify, update, extend, or redeploy applications without affecting other workloads.

These technical benefits bubble up into significant business benefits that improve your competitive advantage, reduce costs, save time, and bolster the bottom line:

- Shorten software's time to market.
- Improve developer agility and productivity.
- Respond faster to change.

BENEFITS FOR DEVELOPERS

The business value of containers and Kubernetes isn't limited to the business as a whole or the office of the CIO. Developers like containers because they make life easier, development more engaging, and work more productive.

- Portability: Containers let developers choose how and where to deploy an app.
- Speed: Containers expedite workflows like testing and speed up iterations.
- CI/CD Pipeline: Kubernetes and containers support continuous integration and continuous deployment.
- Flexibility: Developers can code on their laptops when and where they want with the tools they like.
- The 13th Factor: Containers and Kubernetes are seen as fashionable technologies. Developers are highly motivated to use them.

Introduction to Cloud-Native Architectures and Practices

This chapter briefly surveys some of the application architectures and development processes that underlie cloud-native applications.

Microservices

The digital transformation is driving a shift toward new application architectures. Developing a new application or refactoring an existing one with containers and microservices is often motivated by the following outcomes:

- Extend an application's capabilities more easily
- Add new features faster and easier
- Improve maintainability
- Reduce vulnerabilities
- Make it perform faster or scale better

Microservices, coupled with containers, are increasingly becoming the architectural pattern of choice for developing a new application. A microservices architecture breaks up the functions of an application into a set of small, discrete, decentralized, goal-oriented processes, each of which can be independently developed, tested, deployed, replaced, and scaled. For cloud-native applications, the services often take the form of databases, message queues, key-value stores, tooling, and so forth.

For the software development process, a key outcome of using microservices with containers is continuous integration and continuous delivery. A software developer can modify, test, or scale one part of the application without other developers having to rebuild and redeploy other parts of the application. Running containers on virtual machines also adds a beneficial level of isolation to applications built with microservices. You can isolate a set of services from each other and then group them inside a virtual machine.

Applications built with a microservices architecture, however, do not come without their challenges. Running the application's services in production and at scale requires coordination and the right infrastructure. Trying to develop an application with microservices on a laptop or desktop can hit performance and memory constraints. Even when an application does not use microservices, a laptop might not have enough resources to run a copy of a development environment. Developers can

exploit the capacity of a vSphere software-defined data center to develop and test a containerized application, which is a pre-existing advantage for organizations with vSphere environments who want to begin building and deploying cloud-native applications.

For example, VMware Pivotal Container Service (PKS), which includes a commercially supported distribution of Kubernetes, provide a ready-made platform for building cloud-native and twelve-factor applications on existing vSphere infrastructure. When an application built with micro-services is deployed with PKS, Kubernetes manages the microservices, each of which can reside in its own container for scalability.

With PKS, Kubernetes works with a VMware SDDC to supply persistent storage with VMware vSAN™, micro-segmentation with VMware NSX®, and security mechanisms like role-based access control. After a Kubernetes cluster is provisioned in PKS, it can be managed with high availability, auto-recovery from failure, auto-scaling, upgrades, and monitoring.

Using vSphere and PKS to support a microservices architecture is covered in greater detail in later chapters.

THE BENEFITS OF MICROSERVICES

Coupled with containers, microservices are increasingly becoming the architectural pattern of choice for developing a new application. The architecture breaks up the functions of an application into a set of small, discrete, decentralized, goal-oriented processes, each of which can be independently developed, tested, deployed, replaced, and scaled.

- Increase modularity
- Make apps easier to develop and test
- Parallelize development: A team can develop and deploy a service independently of other teams working on other services
- Support continuous code refactoring to heighten the benefits of microservices over time
- Drive a model of continuous integration and continuous deployment
- Improve scalability
- Simplify component upgrades

Deconstructing the Monolith and Other Use Cases

Although Kubernetes is an excellent system for orchestrating containerized applications built with microservices, Kubernetes can serve other use cases, most notably 12-factor apps. The 12-factor app is a methodology for developing a software-as-a-service (SaaS) application—that is, a web app—and deploying it on a platform as a service (PaaS), such as Pivotal Cloud Foundry.

Transitioning to cloud-native architectures is also a key Kubernetes use case. Even though you might not plan on using microservices in the near future, implementing Kubernetes and the right underlying infrastructure will ease the transition to a microservices architecture when you are ready to take that step.

Kubernetes along with developer-ready infrastructure addresses a lingering problem that undermines many organizations: the monolithic application. It is difficult to modify, scale, and redeploy. Lifting and shifting a monolithic application to containers and Kubernetes opens the door to begin breaking it up into easily modifiable, scalable parts later. Its new packaging in a container also increases its agility and portability now.

Another compelling use case is portability. Kubernetes works across different types of clouds. In other words, the portability of containers combined with the power of Kubernetes gives you cloud independence: You can move the same containerized application among a private cloud, a public cloud, or a hybrid cloud with minimal effort.

Flexibility is an intriguing characteristic of Kubernetes. Although it's not a use case per se, flexibility helps you adapt to unknown use cases in the future. For example, you might think of the current mantra of "delivering applications early and often" as a use case. But as your application matures, you might find that other use cases, such as service discovery, become more important. In other words, once you can successfully fulfill one use case, you might aim for another one. The flexibility and evolving power of Kubernetes can help improve your application development and deployment practices over time.

The engineers working on Kubernetes recognize that the platform's flexibility can address new use cases as they emerge. "In our experience, any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, we expect the Kubernetes API to continuously change and grow," the Kubernetes website says.[6]

---

[6]: Kubernetes.io, The Kubernetes API.

Kubernetes for Cloud-Native and 12-Factor Applications

Kubernetes makes containerized applications work in a manageable way at scale. Recall the second part of the definition of cloud-native applications: They are dynamically orchestrated in such a way that containers are actively scheduled and managed to optimize resource utilization. Kubernetes does exactly that. It orchestrates containers and their workloads to optimize the utilization of the virtual machines and physical servers that make up the nodes in a cluster.

Revisiting the 12 factors from the previous chapter details how Kubernetes streamlines application management. In general, Kubernetes can deploy and run 12-factor apps.

HOW KUBERNETES AND CONTAINERS STREAMLINE APPLICATION MANAGEMENT		
	Factor	**Benefit**
1	Deploy the application many times from one codebase.	Kubernetes can deploy applications with one code base many times by giving a pod a specification that includes a container image reference.
2	Declare and isolate dependencies.	Containers can express dependencies.
3	Store the configuration in the environment, not the code.	You can store aspects of an application's configuration in the Kubernetes environment. For example, the ConfigMaps construct separates configuration artifacts from an image's instructions.
4	Connect to supporting services, such as a database, instead of including it in the code.	Kubernetes lets you deploy supporting services, such as a database, in separate containers and then manages all the containerized components together to ensure availability and performance.
5	Treat build and run as separate stages.	You can, for example, build the application by using Jenkins (a pipeline automation server separate from Kubernetes) and then run the Docker images by using Kubernetes.

6	Run the app as stateless processes.	Kubernetes makes it easy to run stateless applications. Kubernetes allows states to be maintained independently in an etcd data store, for instance, while the application runs. Kubernetes also allows you to attach persistent storage. The spec file defining a Pod, for example, can require a persistent volume; if the Pod goes down, the replacement Pod connects to the same persistent volume.
7	Expose services by using port binding.	Kubernetes includes configuration options for exposing services on ports. In the nginx example YAML file that appeared earlier, the nginx web server was bound to Port 80 and exposed as a service.
8	Scale out by adding concurrent processes.	Kubernetes scales an application by adding more Pods. Kubernetes can use the replication controller, for example, to add multiple Pods at the same time.
9	Ensure durability with disposability.	Containers running in Kubernetes are seen as mutable—they are to be stopped and replaced on demand or on a schedule.
10	Make development and production peers.	The Kubernetes environment lets development and production code be rigorously tested in the same way. For instance, you can use a Kubernetes deployment with two pods, one pod that contains the production environment and another pod that contains the staging environment, which in effect makes staging and production peers. In addition, the environment specified in a container is uniform across development and production environments.
11	Process logs as event streams.	Kubernetes lets you access the standard output of a container so that you can process its output as a data stream with the tool of your choice, such as VMware vRealize® Log Insight™.
12	Run management tasks as one-off processes.	You can schedule a Pod consisting of the application container using a different entry point to run a different process, such as a script to migrate a database.

DevOps

DevOps is a key practice driving the development and deployment of cloud-native applications and 12-factor apps. When developers and IT personnel collaborate on operations to release software early and often, or daily or hourly for that matter, you can see DevOps at work in practices, processes, and automation behind the building, testing, and releasing of software.

DevOps breaks down the organizational barriers between developers and IT operators to align both types of roles behind the common goal of quickly turning ideas and innovations into releasable, maintainable software, often by building and using a pipeline for continuous integration, delivery, and deployment. A culture of collaboration is key, not only between the two types—who often come together to form a single team—but also with other teams and organizations, such as security.

There are common, if not universally accepted, guidelines that underscore DevOps:

- Move the infrastructure for production deployments into the delivery pipeline
- Treat infrastructure as code
- View infrastructure as immutable
- Employ agile methodologies
- Produce small, frequent releases

Profile of a DevOps Engineer: Responsibilities and Skills

Here's an example of what a hypothetical engineer working in a geographically distributed DevOps team might do. In general, the engineer's focus is on building and operating PaaS, SaaS, and on-premise solutions to help the consumers of the solutions manage, govern, and secure applications running in private and public clouds. The DevOps engineer applies innovations from both open-source and proprietary solutions.

Here are some of the DevOps engineer's responsibilities:

- Designing, building, and managing DevOps processes, infrastructure, and tools throughout the CI/CD pipeline. The work includes planning, coding, testing, releasing, deploying, maintaining, and monitoring systems and apps.

- Creating and implementing automation to test and validate software artifacts as they move through the CI/CD pipeline.
- Interacting with developers to make sure that the CI/CD pipeline, automation tests, infrastructure, and tools fulfill requirements for usability, scalability, performance, security, and productivity.

And here are some of the DevOps engineers skills:

- The ability to write Bash scripts and to code in scripting languages like Python.
- A thorough understanding of microservices architectures as well as traditional monolithic architectures. Both skills are useful because legacy apps with monolithic architectures might be refactored with a microservices architecture.
- Experience creating and operating a CI/CD pipeline and its associated processes for SaaS and on-prem applications by using Maven, Git, Gerrit, and Jenkins.
- Hands-on experience with NoSQL databases like MongoDB, Redis, and Cassandra
- Hands-on experience with datastores and libraries for search systems, such as Lucene, Solr, and Elasticsearch.
- Expertise with configuration management tools, including Ansible, Chef, Puppet.
- Experience designing and operating solutions on such cloud platforms as Google, Azure, and AWS.
- Familiarity with logging and monitoring tools like Prometheus and vRealize Log Insight.
- The ability to deploy and operate virtual machines and containers by using cloud-native technology: Vagrant, Docker, Kubernetes.

Continuous Integration and Continuous Deployment

Developers and DevOps engineers use the CI/CD pipeline to develop, commit, integrate, and test code in the process of creating a software artifact that can be automated, configured, deployed, and monitored. Although different teams frequently use different tools, a prototypical CI/DC pipeline might look something like this:

1. Use tools such as JIRA or GitHub Issues to plan a release or changes to a release.

2. Use a tool such as Atom to write code and unit tests in languages like Python, Java, and Go.
3. Commit changes with Git or GitHub.
4. Use tools such as Jenkins and Gerrit to continuously integrate those changes.
5. Use a tool such as vRealize Automation to test code.
6. Create a software artifact by using JFrog Artifactory.
7. Perform continuous deliver by using Jenkins.
8. Manage configuration by using Chef, Ansible, or Puppet.
9. Monitor the application by using vRealize Operations and vRealize Log Insight.

Container Technology in the Software-Defined Data Center

The problems that a software-defined data center (SDDC) addresses stem from the digital transformation reshaping business. Companies of all types are under pressure to create innovative software that engages their customers. The digital technologies at the source of this shift are cloud computing, mobile devices, and data analytics. Companies can exploit these technologies to lower costs, connect with customers, and improve their bottom line.

But reinventing a traditional company, or even a technology company, as a contemporary software-centric enterprise requires the creation of applications that run in the cloud and the infrastructure and tools to build them. To accelerate the development of innovative software and to adapt to changes in the marketplace, you are likely to need such technologies as containers, microservices, distributed systems, orchestration tools, and virtualization.

For their infrastructure, technology-savvy companies seek robust, API-driven solutions that scale to handle large volumes of data at pace. But putting in place scalable, flexible infrastructure that fosters the development and deployment of cloud applications can be complex, difficult, and costly.

The fast track to cost-effectively adopt containers is to transform your existing virtualized infrastructure into a flexible, scalable, modernized data center capable of deploying cloud-native applications as well as continuing to host traditional apps. Enterprises are increasingly moving toward container-based architectures to develop applications faster, increase automation, and reduce server costs.

VMware vSphere plays a central role in this approach.

THE BENEFITS OF MODERNIZING

- Faster development and deployment
- Agility
- Flexibility
- Scalability
- Portability
- Process automation

- Resource optimization
- Easier maintenance
- Automated operations
- Heightened security

VMware vSphere and the SDDC

Enterprises worldwide have used VMware vSphere® to significantly improve IT efficiency and performance, yet the mobile cloud era presents new challenges. To meet this challenge, IT organizations need to virtualize the rest of the data center so all infrastructure services become as inexpensive and easy to provision and manage as virtual machines.

The software-defined data center (SDDC) establishes the ideal architecture for private, public, and hybrid clouds. Pioneered by VMware, SDDC extends the virtualization concepts you know—abstraction, pooling, and automation—to all data center resources and services, including network virtualization and software-defined storage.

At the same time, automated management delivers a framework for policy-based management of data center application and services. The result is unprecedented IT agility and efficiency, with flexibility to support a range of hardware and applications. As a result, infrastructure utilization and staff productivity increase, substantially reducing both capital expenditures and operating costs.

Abstract and Automate: Network Virtualization

VMware NSX provides network virtualization for an SDDC, abstracting Layer 2 through Layer 7 networking functions—such as switching, firewalling, and routing—on top of your existing physical network. NSX embeds the networking and security functionality typically handled by hardware directly in the hypervisor. NSX creates what can be thought of as a "network hypervisor" that is distributed throughout the data center.

Virtualization thus becomes the operational model for networking and security, unlocking the ability for IT to move at the speed of business. By moving network and security services into the data center virtualization layer, network virtualization enables IT to create, snapshot, store, move, delete, and restore entire application environments with the same simplicity and speed that they now have when spinning up virtual machines.

This abstraction, in turn, enables levels of security and efficiency that were previously infeasible. IT is empowered to become an enabler of innovation for the organization. IT can help multiple stakeholders instead of treating their requests as competing and mutually exclusive.

IT can, for example, apply micro-segmentation with distributed stateful firewalling and dynamic security policies attached directly to individual workloads. Micro-segmentation is defined and discussed in detail later in this chapter.

Risk-Free Scale Out with Ease: Virtual Storage

As the only vSphere-native software-defined storage platform, VMware vSAN accelerates the modernization of infrastructure by delivering an agile solution ready for next-generation applications. vSAN seamlessly extends virtualization to storage with a secure, flash-optimized solution that integrates with the the VMware ecosystem to handle critical workloads running at cloud scale.

vSAN is built on industry-standard x86 servers and components that help lower TCO by up to 50 percent compared with traditional storage solutions. vSAN pools together server-attached storage to provide a highly resilient shared datastore suitable for any virtualized workload, including cloud-native applications and DevOps infrastructure. By being tightly integrated with the vSphere kernel, vSAN sits directly in the I/O data path, where it can optimize the data I/O path with high performance and minimal impact on CPU and memory.

For a software-defined data center, vSAN gives you granular non-disruptive scale-up or scale-out store so that you can expand capacity and performance by adding hosts to a cluster (scale-out) or expand only capacity by adding disks to a host (scale-up).

Also in keeping with a key feature of an SDDC, vSAN includes VM-centric policy-based management: vSAN uses storage policies, applied on a per-VM basis, to automate provisioning and balancing of storage resources.

vSAN also includes built-in failure tolerance and advanced availability. More specifically, vSAN leverages distributed RAID and cache mirroring to ensure that data is never lost if a disk, host, network, or rack fails. vSAN seamlessly supports vSphere availability features like vSphere Fault Tolerance and vSphere High Availability. Using vSAN for storing the data of cloud-native applications is discussed later in the book.

Put a Lid on It: Security for Containers

Security poses an obstacle to container adoption. The increased attack surface of containers and other factors can heighten risk. Running containerized applications on virtual machines, however, decreases the attach surface of containers and lowers risk.

Heightening Security by Running Containers on VMs

In September 2017, the National Institute of Standards and Technology published *Application Container Security Guide*, which is also known as NIST Special Publication 800-190. It explains the potential security concerns surrounding the use of containers and sets forth recommendations for addressing these concerns.

The National Institute of Standards and Technology (NIST) is a U.S. federal technology agency working with industry to develop and apply technology, measurements, and standards. NIST works with standards bodies to create international cybersecurity standards.

An important implication of the *Application Container Security Guide* is to run containerized applications on virtual machines. "While containers provide a strong degree of isolation, they do not offer as clear and concrete of a security boundary as a VM. Because containers share the same kernel and can be run with varying capabilities and privileges on a host, the degree of segmentation between them is far less than that provided to VMs by a hypervisor."[7]

One conern of the *Application Container Security Guide* is that containers or the operating system of a physical host can easily be misconfigured, increasing the attack surface and the level of risk. In contrast, the abstraction, automation, and isolation of an operating system running on a virtual machine in a hypervisor environment reduces the attack surface and decreases the risk of a security breach. "Carelessly configured environments can result in containers having the ability to interact with each

[7] NIST Special Publication 800-190, *Application Container Security Guide*, by Murugiah Souppaya, Computer Security Division Information Technology Laboratory; John Morello, Twistlock, Baton Rouge, Louisiana; Karen Scarfone, Scarfone Cybersecurity, Clifton, Virginia. September 2017. This publication is available free of charge from https://doi.org/10.6028/NIST.SP.800-190

other and the host far more easily and directly than multiple VMs on the same host," the *Application Container Security Guide* says.[8]

Virtual machines and containers should be seen as complements, not substitutues. "Although containers are sometimes thought of as the next phase of virtualization, surpassing hardware virtualization, the reality for most organizations is less about revolution than evolution. Containers and hardware virtualization not only can, but very frequently do, coexist well and actually enhance each other's capabilities, the *Application Container Security Guide* says. "VMs provide many benefits, such as strong isolation, OS automation, and a wide and deep ecosystem of solutions. Organizations do not need to make a choice between containers and VMs. Instead, organizations can continue to use VMs to deploy, partition, and manage their hardware, while using containers to package their apps and utilize each VM more efficiently."

Securing the Orchestration System

Another concern of the *Application Container Security Guide* is recommending countermeasures to secure the orchestration system managing containers. The suggested countermeasures in the guide include the following:

- Granular access control of administrative actions based on hosts, containers and images as parameters.
- The use of enterprise-grade authentication services using strong credentials and directory services.
- Isolating containers to separate hosts based on the sensitivity level of the applications running in them.

Another NIST document, *Security Assurance Requirements for Linux Application Container Deployments*, sets forth security requirements and countermeasures to help meet the recommendations of the *Application Container Security Guide* when containerized applications are deployed in production environments. The orchestration system or its components and tools should meet the following capabilities[8]:

- Logging and monitoring of resource consumption of containers to ensure availability of critical resources.
- The orchestration system must work with many container hosts,

[8] NIST.IR 8176, *Security Assurance Requirements for Linux Application Container Deployments,* by Ramaswamy Chandramouli, Computer Security Division, Information Technology Laboratory. October 2017. This publication is available free of charge from https://doi.org/10.6028/NIST.IR.8176

not just one, to be able to provide a global summary of resource usage for all running containers.

Micro-Segmentation for Containerized Workloads

Micro-segmentation uses network virtualization to divide a data center and its workloads into logical segments, each of which contain a single workload. You can then apply security controls to each segment, restricting an attacker's ability to move to another segment or workload.[9]

From this basic definition, you can see that for a data center, micro-segmentation reduces the risk of attack, limits the damage from an attack, and improves security. According to *VMware NSX Micro-segmentation Day 1*, the micro-segmentation capabilities of VMware NSX can implement the following security controls:[10]

- Distributed stateful firewalling, which can protect each application running in the data center with application level gateways that are applied on a per-workload basis.
- Topology agnostic segmentation, which protects each application with a firewall indepedent of the underlying network topology.
- Centralized ubiquitous policy control of distributed services, which controls access with a centralized management plane.
- Granular unit-level controls implemented by high-level policy objects, which can create a security perimeter for each application without relying on VLANs.
- Network-based isolation, which implements logical network overlays through virtualization.
- Policy-driven unit-level service insertion and traffic steering, which can help monitor network traffic.

NIST Special Publication 800-125B, *Secure Virtual Network Configuration for Virtual Machine (VM) Protection*, sets forth recommendations for securing virtualized workloads. The micro-segmentation capabilities of NSX satisfy the security recommendations made by NIST for protecting virtual machine workloads. For more information, see *VMware NSX Micro-segmentation Day 1*.

---

[9] For more information about what micro-segmentation is and what is isn't, see *Micro-segmentation for Dummies*, by Lawrence Miller and Joshua Soto, published by John Wiley & Sons, Inc. 2015.

[10] *VMware NSX Micro-segmentation Day 1,* by Wade Holmes, published by VMware Press, 2017.

Linux Container Hosts

Linux distributions built specifically for running containers are another piece of the cloud-native puzzle and the modernized data center. These container-specific operating systems typically minimize the number of packages, components, and tools they include, focusing instead on providing just-enough of an operating system for efficiently and securely running containerized applications. Container-specific operating systems include Google Container-Optimized OS, CoreOS Container Linux, Project Atomic, and Project Photon OS.

This section looks at an example of a Linux container host—Photon OS—to explain the part it plays in solving a range of problems that would otherwise limit the deployment of cloud-native applications.

Photon OS Overview

Project Photon OS™ is an open source Linux container host optimized for cloud-native applications, cloud platforms, and VMware infrastructure. Photon OS provides a secure runtime environment for running containers.

By minimizing the number of packages, focusing on security, and providing advanced lifecycle management, Photon OS efficently runs containers on VMware vSphere, Microsoft Azure, Google Compute Engine, and Amazon Elastic Compute Cloud.

Photon OS comes in a minimal version and a full version. Each version contains only the elements necessary to fulfill its use case. The minimal version is a lightweight host tailored to running containers when performance is paramount. The full version of Photon OS includes additional packages to help develop, test, and deploy containerized applications. Both versions of Photon OS yield several benefits:

- An improvement in resource-efficiency by using smaller server builds
- A reduction in security risks by removing vulnerable components
- A decrease in management effort by having fewer components to update

Photon OS includes the open source version of Docker to streamline the workflow of getting a container running in a hypervisor. A developer can install a hypervisor such as VMware Fusion® on a laptop, replicate a cluster of virtual machines, and then, with Photon OS, build containerized applications.

Security-Hardened Linux

The design of Photon OS prioritizes security. Photon OS secures itself with its build process, compiler settings, root password rules, and PGP-signed packages and repositories. A system administrator or DevOps manager can enforce security with vulnerability scans, the pluggable authentication modules, the Linux auditing service, and many other measures.

Photon OS also works with VMware's open source Lightwave project to set up a certificate store and a secure LDAP directory service. Lightwave also integrates Photon OS machines with Microsoft Active Directory or LDAP for authentication and access control. See the section on Lightwave.

As a streamlined Linux operating system that the Photon OS team compiles from source, Photon OS is hardened in part by its build process. The Photon OS team can audit packages, such as OpenSSL, to identify vulnerabilities before releasing the system. Vulnerabilities can be fixed by applying and testing security patches as soon as they become available.

Life-Cycle Management

Photon OS seeks to reduce the burden and complexity of managing clusters of Linux machines by including an efficient packaging model, extensibility, and centralized administration in its fundamental design. Here are some of Photon's design elements that simplify life-cycle management:

- Atomic updates with RPM-OSTree
- Incremental stateful updates (RPMs)
- Package repositories curated by VMware
- Extensible distribution: You can add and remove functionality incrementally
- Signed repositories

These design elements come together to make it easy to update the system, perform in-place upgrades, and refresh installed packages like Docker and Kubernetes. Because the minimal version of Photon OS is designed to be mainly a read-only OS, it can be replaced in an atomic fashion.

Photon OS performs incremental stateful updates of RPMs. As a DevOps manager, you can update an application such as Docker individually without having to update an entire branch, as some other operating systems require you to do.

The RPM-based approach to managing the operating system makes Photon OS extensible. As an extensible RPM-based distribution, you can add or remove applications individually. For example, you can take the minimal version and then just install postgres if that is all the support your application needs.

Moving Containerized Applications from Development to Production

By integrating seamlessly with the VMware ecosystem—including VMware vSphere—Photon OS delivers a ready-made foundation for rapidly building and deploying cloud-native applications while continuing to fulfill such IT requirements as cost-effectiveness, performance, and security.

Because Photon OS is optimized to work with VMware vSphere, VMware Workstation Pro™, and VMware Fusion, Photon OS empowers you to seamlessly migrate container-based applications from development to production while—unlike other systems—helping to maintain the isolation and security of the application running in the container.

Running applications in containers on Photon OS machines integrated with vSphere overcomes a significant problem that plagues containerized applications: They are difficult to deploy into production securely. The security of virtual machines running in vSphere coupled with the security-first design of Photon OS and your own network security measures helps establish production-level security for the containerized application.

There is another hurdle that keeps developers from deploying containers in production: IT operations. At many companies, developers and IT are separate entities. Without the cooperation and commitment of IT, developers can do little to move their Docker workloads into production. By tying in seamlessly with VMware vSphere, which often forms the basis of the production environment that is owned and operated by IT, Photon OS paves the way to put containerized applications into production. Photon OS takes this connection with virtual infrastructure one step further—and helps alleviate IT concerns over security—by integrating with VMware's Project Lightwave™, a security suite for cloud-native platforms.

KEY PHOTON OS PROPERTIES

Photon OS is a lightweight Linux operating system optimized for running containers in a software-defined data center.

- Optimized kernel: The Linux kernel is tuned for performance when Photon OS runs on VMware ESXi™ and VMware vSphere.
- Docker ready: The Docker daemon is included in the distribution to ease running containers.
- Security-hardened Linux: The kernel is configured according to the recommendations of the Kernel Self-Protection Project (KSPP).
- Curated packages and repositories: Packages are built with hardened security flags.
- Secure EFI boot: The operating system boots with validated trust.
- Secure remote management: The Photon Management Daemon securely manages the firewall, network, packages, and users on remote Photon OS machines by using API calls over a command-line utility, Python, or REST.
- Support for persistent volumes: Photon OS supports persistent volumes to store the data of cloud-native apps on VMware vSAN™.
- Project Lightwave™ integration: Lightwave is an open source security platform from VMware that authenticates and authorizes users with Active Directory or LDAP.
- Advanced lifecycle management: There are timely security patches and updates to container packages, such as Docker and Kubernetes.

Securing Cloud Platforms with Lightwave

The need for security in the cloud is acute. Extending the security frameworks, standards, and policies of your on-premises infrastructure to your resources in the cloud establishes the level of consistency that's required to protect the integrity, availability, and confidentiality of your cloud-native operations.

To consistently apply your on-premises security controls and policies in the cloud, several requirements for high-quality identity and access management come to the fore:

- Standards
- Flexibility, portability, and cloud-platform independence

- Interoperability
- Scalability
- Administrative control

Standards are a key requirement because they let you apply trusted tools and protocols across disparate environments to reduce the risk of security incidents and compliance problems. Flexibility enables you to port your security policies and controls from one environment to another as you move a server or application. As more workloads migrate to the cloud, the cloud-platform independence of your identity service helps you move from one cloud provider to another without having to redeploy or reconfigure identity management systems.

Interoperability ensures that security mechanisms are compatible with other systems. Scalability addresses the need for cloud-scale as operations expand. And administrative control empowers you to implement the security frameworks and policies that you want while reducing your reliance on cloud providers and third parties.

Security Problems in Cloud Computing

The multitenant environment of public clouds complicates identity and access management. In the cloud, it is important to securely authenticate system users and administrators, giving them access only to the resources they own or need to do their jobs. But authentication and access control become difficult when assets are spread across both on-premises data centers and cloud services. Porting identities and access policies to the cloud depends on how easily you can integrate your corporate identity directories and policies with the service provider's systems.

Implementing seamless security across both your on-premises environment and multiple public clouds is increasingly becoming a necessity. Project Lightwave™ is a massively scaled, multitenant, open-source identity platform from VMware that solves this problem by delivering a standards-based directory service, Active Directory integration, certificate services, and Kerberos authentication.

LIGHTWAVE SERVICES

- Identity management and directory services
- Authentication and authorization
- Certificates

Implementing Cloud-Scale Security with Lightwave

Lightwave meets the requirements of these use cases with its directory service, Active Directory interoperability, Kerberos authentication, and certificate services. Lightwave provides the following services:

- Directory services and identity management with LDAP and Active Directory interoperability
- Authentication services with Kerberos, SRP, WS-Trust (SOAP), SAML WebSSO (browser-based SSO), OAuth/OpenID Connect (REST APIs), and other protocols
- Certificate services with a certificate authority and a certificate store

Using these Lightwave security services in the cloud empowers IT security managers to impose the proven security policies and best practices of on-premises computing systems on their cloud computing environment. The Lightwave security services work in the cloud, in an on-premises data center, and in a hybrid cloud. The security frameworks, standards, and policies that come with Lightwave can follow users, applications, and workloads nearly anywhere.

Directory Services and Identity Management

Lightwave is an extensible identity platform that works with multiple identity sources, including Microsoft Active Directory, LDAP, OpenLDAP, and MIT Kerberos with LDAP. The platform includes a REST API for LDAP, integrated DNS, and support for the System for Cross-domain Identity Management (SCIM). SCIM is an open standard that simplifies identity management in the cloud by automatically exchanging user identities with a REST API between domains.

At the core of Lightwave is a standards-based, AD-compatible LDAP 3 directory service with multimaster replication. The LDAP service, which can be managed with LDAP-compliant browsers, supports such operations as bind, add, modify, delete, search, extended operation, and controls. It manages users and groups, including nested groups, and provides policy-based password management.

The directory service uses the following secure data access mechanisms:

- Generic Security Service Application Program Interface (GSSAPI) over the Secure Remote Password protocol (SRP)
- Simple Authentication and Security Layer (SASL) over SRP

Designed for multitenancy, the directory service includes a hierarchical directory store that can accommodate multiple tenants at scale. A directory information tree (DIT) isolates the data of each tenant by placing each tenant in its own subtree. The ACL for a tenant's subtree is for only the tenant's own administrator. Each entry under a tenant gets its own ACL that is based on a security descriptor; each object, that is, receives its own ACL. Lightwave includes a tool for browsing and editing entries in the directory.

Scalability and Performance

For scalability, the directory service includes an extensible LDAP schema, an active-active multimaster scheme, and dynamic indexing. For performance, the directory service uses a Lightning Memory-Mapped Database (LMDB). It is an ACID-compliant persistent data store from OpenLDAP that has the following performance-enhancing features:

- B+ tree key-value store
- Single writer plus many readers
- Multi-version concurrency—the readers never block the writer
- Memory-mapped file with copy-on-write
- Write-ahead logging developed by VMware

Replication

For replication, the directory service uses a state-based scheme for eventually consistent multi-node LDAP replication. Every directory node in a Lightwave domain accepts write requests. On Lightwave, the replication service includes a tool with a user interface and a single command to add or remove a node, which simplifies topology management. In addition, backup and restore is supported on a per-node basis. Overall, this approach to replication simplifies the life-cycle management of a domain.

Architecture

The following architectural diagram summarizes the main components of the Lightwave directory service.

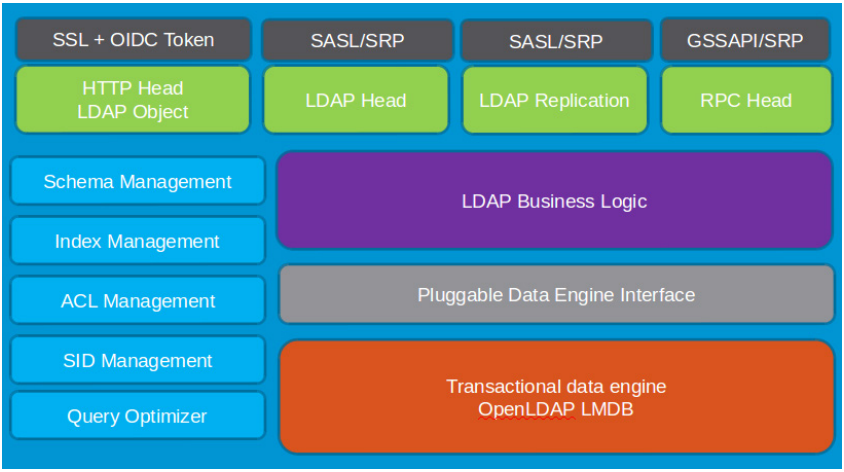

Figure 1: The architecture of the Lightwave directory service.

Authentication Services

The authentication services of Lightwave contain two main components: A server that acts as a Kerberos 5 key distribution center and a secure token service that supports the OIDC, WS-TRUST, and SAML 2.0 (WebSSO) standards for single sign-on.

In Lightwave's converged identity model, Kerberos, OAuth 2.0, and OIDC are integrated with the LDAP directory server process.

The secure token service can issue Security Assertion Markup Language (SAML) 2.0 tokens as well as OIDC tokens. Lightwave can be integrated with Active Directory to issue secure tokens to principals defined in Active Directory forests. With SAML 2.0, Lightwave gives a user SSO access to different services by using the same credentials.

Lightwave works with OAuth 2.0 and OpenID Connect, a protocol for authentication and authorization that lets you set up one SSO service for different cloud services. After users enter their credentials to access one service, they don't need to do it again to access others.

Lightwave also works with the WS-Trust standard to issue and validate security tokens and to broker trust relationships between parties exchanging secure messages.

Kerberos Key Distribution Center

The Kerberos key distribution center (KDC) handles authentication and authorization for Kerberized applications. LDAP, SASL, DCE/RPC, and GSSAPI applications can all use Kerberos. Security principals are stored in a replicated directory, and Kerberos service tickets are extended to include Lightwave authorization data.

Features and Capabilities

The secure token service supports the WS-Trust, SAML, OAuth2 and OpenID Connect standards with the following capabilities:

- Browser-based single sign-on (SSO)
- Full multi-tenancy support
- External SAML Federation
- Just-in-time provisioning (JIT)
- IDP discovery and selection
- Multiple identity sources, including the native VMware directory, Microsoft Active Directory, and OpenLDAP
- Full Kerberos support and full AD domain trust support for integrating authentication with Microsoft Windows and Active Directory
- Schema customization for OpenLDAP to support a broad range of OpenLDAP deployments
- Two-factor smart card support with a common access card (CAC) or with an RSA SecurID token
- REST management APIs

COMPONENTS IN THE LIGHTWAVE ARCHITECTURE

To summarize, the Lightwave security services form an architecture that comprises these components:

- The directory service and its directory store
- A Kerberos key distribution center that is integrated with the directory service
- An integrated DNS server
- A multi-protocol secure token service (STS) for authentication and authorization
- A certificate authority
- A certificate store

Certificate Services

The Lightwave certificate service includes an X509-compliant certificate authority and a certificate store. The certificate authority issues and revokes certificates, and the certificate store holds certificates and keys. Together they provide a complete certificate management stack that integrates with LDAP and establishes user identities with Kerberos authentication.

The Lightwave Certificate Authority

The certificate authority issues signed X.509 digital certificates and supports the PKIX standard. It can distribute CA roots and CRLs over HTTP and LDAP in accordance with RFC 4387. It also supports CSR and key generation as well as auto-enrollment and certificate revocation. Secured and authenticated by Kerberos, the certificate authority validates certificate requests by analyzing key usage, extensions, SAN, and other factors. Server policies can be used to automatically approve or reject certificates. The certificate authority has a dual mode in which it can act as an enterprise root CA or as a subordinate or intermediate CA. The key lengths are strong, ranging from 1 K to 16 K, and the hashing algorithms use SHA-1 or SHA-2, the latter of which is the default. The key usage is encryption and signing. The certificate file formats are PKCS12, PEM and JKS.

For administration, Lightwave lets you access the certificate authority with a user interface or command-line utilities, including diagnostic tools. It also supports certificate-auditing requirements.

The Lightwave Certificate Store

The Lightwave endpoint certificate store holds certificates, private keys, and certificate revocation lists (CRLs). Lightwave controls access to the certificate store by using Kerberos. By default, only the user who created the store—that is, the owner—has access. The certificate store typically contains three kinds of entries:

- A private key associated with a certificate or certificate chain
- A certificate of a trusted entity
- A certificate revocation list published by the Lightwave certificate authority

Lightwave in vSphere and vCenter

Lightwave acts as the directory service, authentication engine, secure token service, lookup service, certificate authority, and certificate store in deployments of VMware vCenter® and VMware vSphere® 6. In addition, Lightwave lets system administrators join a vCenter instance to Active Directory. In vSphere 6, the Lightwave components are collectively known as the VMware Platform Services Controller (PSC).

It handles such security functions as single sign-on and certificate management. vCenter provides an example of how Lightwave delivers single sign-on to an enterprise platform. When a user authenticates with the Lightwave identity management service on vCenter, the user receives a SAML token issued by the embedded Lightwave secure token service. With the SAML token, the user can then use any vCenter service (and perform actions the user has privileges for) without having to sign in again. The vCenter single sign-on service signs tokens with a signing certificate and stores the token-signing certificate. The service's certificate is also stored.

As you will see in later chapters, understanding the security components and capabilities of Lightwave is important because Lightwave plays a fundamental underlying role in securing VMware platforms that use vSphere to deliver containers and Kubernetes as a service. Using Lightwave security services in the cloud empowers IT administrators and DevOps managers to impose the proven security policies and best practices of on-premises computing systems on their cloud computing environment. The security standards and protocols that come with Lightwave can work across clouds, on-premises data centers, and hybrid clouds.

Managing Container Images with Harbor

This section looks at a secure container registry called Harbor to shed light on the role a registry plays in a modernized data center.

Project Harbor is an open source, enterprise-class registry server from VMware that stores and distributes Docker images in a private registry behind your firewall. Harbor extends the open source Docker distribution by adding such functionality as security and management. Harbor can be set up with multiple registries, and images can be replicated across the registries. Harbor provides a graphical portal, shown in the following image, and a RESTful API for managing repositories. Images are protected with role-based access control.

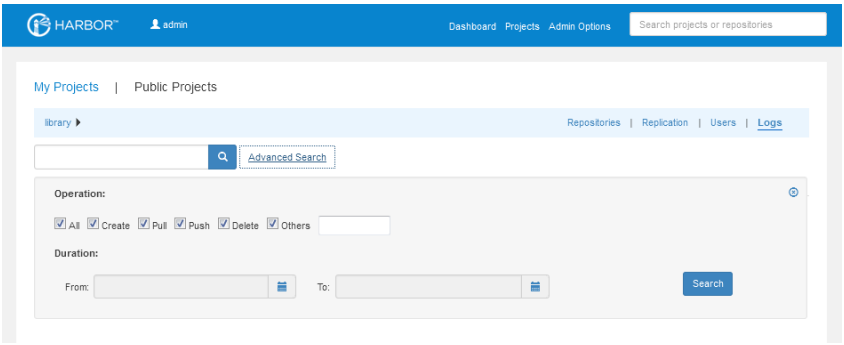

Figure 2: The graphical user interface of the Harbor portal.

For security, Harbor integrates with Project Lightwave, Active Directory, or LDAP to authenticate users. Harbor also tracks user interactions for auditing.

Use Cases

- Enterprise-wide use of container technology.
- Secure container images for use in production.
- Extend Docker with management and security, including integration with identity management systems.
- Manage access to container images with Microsoft Active Directory, LDAP, or Lightwave.

Key Features and Benefits of Harbor

By placing registries closer to the build-and-run environment, Harbor improves image transfer efficiency. It also supports the setup of multiple registries and replicates images between them. Storing images within the private registry keeps data behind the enterprise firewall. In addition, Harbor offers advanced security features, such as user management, role-based access control (RBAC), notary, and activity auditing.

- Image replication images can be synchronized between multiple registry instances, aiding load balancing, availability, and flexibility of adoption.
- Graphic user portal: enables browse and search functions for Docker repository management.

- Role-based access control: users can be added to specific projects with varying levels of permission.
- AD/LDAP support Harbor integrates with existing enterprise AD/LDAP for user authentication and management.
- Image deletion and garbage collection images can be deleted and their space recycled.
- Auditing all repository operations are tracked.
- Notary image authenticity is ensured.
- Internationalization: localized for English and Chinese.
- RESTful API provided for most administrative operations of Harbor, easing integration with other management software.
- Easy deployment: installable both online and offline.
- Secure container images with role-based access control, integration with identity management systems, and vulnerability scanning.
- Manage images and repositories with an intuitive, dynamic user interface.

Component Architecture

Here is a diagram that depicts the architecture of Harbor.

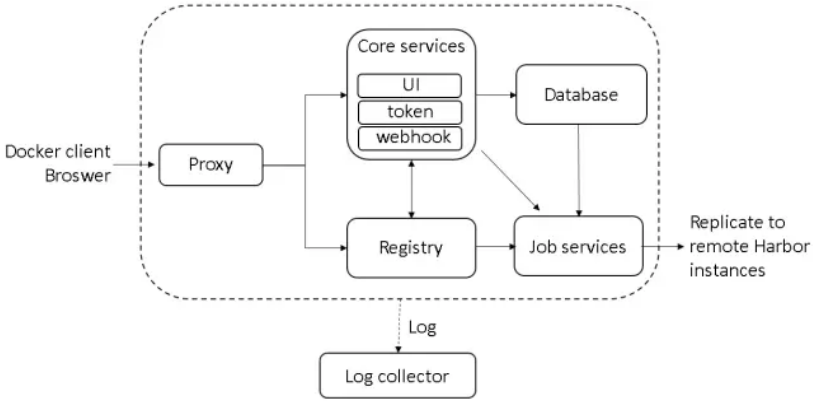

Figure 3: The component architecture of Harbor.

Proxy: The components of Harbor, such as the registry, UI, and token services, are all behind a reversed proxy. The proxy forwards requests from browsers and Docker clients to the backend services.

Registry: The registry stores Docker images and processes Docker push and pull commands. Because Harbor enforces access control to images, the registry directs clients to a token service to obtain a valid token for each pull or push request.

UI: The graphical user interface helps you manage images on the registry webhook, which is a mechanism configured in the registry to populate changes in the status of images. Harbor uses the webhook to update logs, initiate replications, and perform some other functions.

Token service: The token service issues a token for every docker push or pull command.

Database: The database stores the metadata of projects, users, roles, replication policies, and images.

Job services: The job services replicates and synchronizes images across instances of Harbor.

Log collector: It collects the logs of other modules in a single place.

Implementation

Each component of Harbor is wrapped as a Docker container. Naturally, Harbor is deployed by Docker Compose. In the source code (https://github.com/vmware/harbor), the Docker Compose template used to deploy Harbor is located at /Deployer/docker-compse.yml. Opening this template file reveals the six container components making up Harbor.

proxy: Reverse-proxy formed by the Nginx Server.

registry: Container instance created from the official image of Docker distribution.

UI: Core services within the architecture. This container is the main part of Project Harbor.

MySQL: Database container created from the official MySql image.

job services: Replicating images to a remote registry via state machines. Image deletion can also be synchronized to a remote Harbor instance.

log: Container that runs rsyslogd, used for collecting logs from other containers through the log-driver mode.

These containers are linked through DNS service discovery in Docker. By this means, each container can be accessed by their names. For the end user, only the service port of the proxy (Nginx) needs to be revealed.

Component Interaction

The following examples of Docker commands illustrate the interaction among Harbor's components.

The process of docker login

Suppose Harbor is deployed on a host with the following IP address: IP 192.168.1.10. A user runs the docker command to send a login request to Harbor:

```
$ docker login 192.168.1.10
```

After the user enters the required credentials, the Docker client sends an HTTP GET request to the address "192.168.1.10/v2/". The different containers of Harbor will process it according to the following steps:

(a) First, this request is received by the proxy container listening on port 80. Nginx in the container forwards the request to the Registry container at the backend.

(b) The Registry container has been configured for token-based authentication, so it returns an error code 401, notifying the Docker client to obtain a valid token from a specified URL. In Harbor, this URL points to the token service of Core Services.

(c) When the Docker client receives this error code, it sends a request to the token service URL, embedding username and password in the request header according to basic authentication of HTTP specification.

(d) After this request is sent to the proxy container via port 80, Nginx again forwards the request to the UI container according to pre-configured rules. The token service within the UI container receives the request, it decodes the request and obtains the username and password.

(e) After getting the username and password, the token service checks the database and authenticates the user by the data in the MySql database. When the token service is configured for LDAP/AD authentication, it authenticates against the external LDAP/AD server. After a successful

authentication, the token service returns a HTTP code that indicates the success. The HTTP response body contains a token generated by a private key.

At this point, one docker login process has been completed. The Docker client saves the encoded username/password from step (c) locally in a hidden file.

The Process of docker push

After the user logs in successfully, a Docker image is sent to Harbor via a Docker Push command:

```
# docker push 192.168.1.10/library/hello-world
```

(a) Firstly, the docker client repeats the process similar to login by sending the request to the registry, and then gets back the URL of the token service;

(b) Subsequently, when contacting the token service, the Docker client provides additional information to apply for a token of the push operation on the image (library/hello-world);

(c) After receiving the request forwarded by Nginx, the token service queries the database to look up the user's role and permissions to push the image. If the user has the proper permission, it encodes the information of the push operation and signs it with a private key and generates a token to the Docker client;

(d) After the Docker client gets the token, it sends a push request to the registry with a header containing the token. Once the Registry receives the request, it decodes the token with the public key and validates its content. The public key corresponds to the private key of the token service. If the registry finds the token valid for pushing the image, the image transferring process begins.

Integration with Kubernetes

Here's how to deploy Harbor on Kubernetes. This section assumes you know the following aspects of Kubernetes work: Replication Controller, Service, Persistent Volume, Persistent Volume Claim, Config Map.

First, you need to download the docker images of Harbor.

1. Download the offline installer of Harbor from the release page.

2. Uncompress the offline installer and get the images tgz file harbor.*.tgz.
3. Load the images into docker by running the following command: docker load -i harbor.*.tgz

Second, you need to configure it.

A Python script make/kubernetes/prepare is provided to generate Kubernetes ConfigMap files. The script is written in Python, so you need a version of Python in your deployment environment. Also the script needs openssl to generate private key and certification, so make sure you have a workable openssl.

For more information and a link to the release page, see Integrate with Kubernetes at https://github.com/vmware/harbor/blob/master/docs/kubernetes_deployment.md.

Integration with VMware vSphere Integrated Containers for Improved Security

Enterprise private container registry: With Harbor, vSphere Integrated Containers offers an enterprise private container registry with advanced security features such as identity management, LDAP integration, role based access control, and trusted content, all of which help ensure security for container images. With the private registry, you can furnish project-level content trust and notary services to container images. Vulnerability scanning helps prevent vulnerable container images from running in your data center.

More Info on Project Harbor

Download the installer from the https://github.com/vmware/harbor/releases. Then see the Installation and Configuration Guide at https://github.com/vmware/harbor/blob/master/docs/installation_guide.md for intructions.

After installation, see the user guide at https://github.com/vmware/harbor/blob/master/docs/user_guide.md.

Providing Persistent Storage

A lack of persistent storage can stand in the way of container adoption, especially for stateful, data-intensive applications. A data-intensive

containerized application requires a robust, elastic, and programmable storage infrastructure with the same level of security, data integrity, high availability, and storage services that are used for traditional applications and IT infrastructure.

Although it is relatively easy to run stateless microservices using container technology, stateful applications require slightly different treatment. Multiple factors need to be considered when containerized applications call for persistent data:

- Because containers are ephemeral by nature, the data needs to persist after a container is restarted or rescheduled.
- When a container is rescheduled, it can start on a different host than the one on which it had been running, and the storage must be made available on the new host so that the container can start quickly and gracefully.
- You should not have to worry about whether the volume and data will be available for a containerized application. The underlying infrastructure should handle the complexity of unmounting and mounting a volume. Some applications, such as Kafka and Elasticsearch, can have strict identity and ID requirements; if a container with a certain ID gets re-scheduled, the the disk associated with that ID must be re-attached to the new container instance.

Project Hatchway

Project Hatchway, an open source storage project from VMware, provides storage infrastructure options for containers in vSphere environments, including hyper-converged infrastructure (HCI) with by VMware vSAN.

By integrating with Kubernetes, Hatchway lets developers consume storage infrastructure as code, abstracting complexity of the underlying storage infrastructure.

Through Project Hatchway, data services such as snapshot, cloning, encryption, de-duplication, and compression are available at the level of a container volume.

Storage for Stateful Apps in Kubernetes

Stateful containers orchestrated by Kubernetes can also use the storage options of vSphere—vSAN, VMFS, NFS, and VVol—with Kubernetes persistent volume, dynamic provisioning, and StatefulSet primitives.

With Hatchway and vSpherre, storage policy-based provisioning of persistent volumes enables applications to specify SLAs and quality of service at the granularity of container volumes. Database workloads scale on demand as a result of the tight integration with the Kubernetes scheduler and features like StatefulSet.

Microservices Meets Micro-segmentation: Delivering Developer-Ready Infrastructure for Modern Application Development with NSX

Customer benefits abound when microservices meet micro-segmentation. At its core, developer-ready infrastructure is about dealing with the practical realities and complications of making a modern developer application development platform (like Pivotal Cloud Foundry) work in harmony with a modern enterprise private cloud (like a VMware SDDC).

Over the last six to nine months, the container ecosystem has really woken up to the production challenges of using any of the leading container frameworks in production. We have discussed this topic on the VMware cloud-native blog recently in relation to Kubernetes and Docker.

Last month, with the introduction of Pivotal Cloud Foundry 1.10, both the VMware NSX team and the Pivotal team shared some initial concepts around developer-ready infrastructure. Both are good reads if you are itching for some technical details.

Developer-ready infrastructure is about removing human bottlenecks from the interaction between developers and IT. The result for the customers of VMware is better products and services, delivered faster than ever to their customers while continuing to meet operational goals of efficiency, security, and reliability.

Let's drill down on that last paragraph. In our customers' environments, the "business" drives much of the overall operating envelope, guidelines and rules that developers and IT alike must adhere to. The problem is that manual intervention is involved at each step of the way, leading to errors, inconsistencies, delays, and inefficiencies.

Before an application change or update happens, there is usually a business requirement or reason that sets the developer in motion. From there, the developer gets to work to modify the application to address the business requirements. Let's call that developer intent. At that point, IT gets the hand off and does the work needed to make that application

update “live” for the customers of the business. You can refer to this as infrastructure response. At some point down the road, for most of our enterprise customers, there is some sort of audit or compliance check that they must adhere to and pass or they will get fined. We will call that audit readiness.

The process seen in most customer environments today is typically a one-way process with manual human-driven actions along each step of the way. When you consider the magic and automation possible with modern application development practices and then you try apply those practices in the real world where terms like “manual,” “one-way,” and “human-driven” are regularly used to describe the current state of most organizations application release and operations model, the helium is removed from the proverbial balloon.

Developer-ready infrastructure puts the helium back in the microservices balloon by allowing organizations to remove the one-way, human-driven process between developers and IT. Let’s take a specific example. With Pivotal Cloud Foundry, developer intent is captured in the application metadata produced when developers create or modify an application. That application metadata can then be used by VMware NSX to automatically program the infrastructure response in the form networks segments, load balancer configuration and firewall permissions. Once programmed, both the infrastructure configuration and the application metadata can be queried at a moment’s notice (audit readiness) to satisfy a compliance check on the business.

Developer-ready infrastructure radically reduces manual infrastructure processes and developers handling non-development tasks resulting in increased developer productivity. It provides secure, software-based compute, storage, networking, and operational tooling optimized for microservice-based application workloads running in containers.

Combining a VMware SDDC with Pivotal’s cloud-native application platform enables developers to deploy the right software, faster and more frequently by eliminating the drag of traditional operational concerns, delays, and extra code to guard against infrastructure issues. Beyond the Pivotal Cloud Foundry and VMware NSX integration, the entire VMware SDDC portfolio is evolving to better support the needs of modern application development platforms.

There is another problem: How do you automate the interaction between business requirements and developer intent? That, too, is a manual, error prone process. For platform deployment and operations, a large part of the solution to that problem is BOSH.

BOSH

BOSH is an open source tool that enables deployment and lifecycle management of distributed systems. It is the primary method used to deploy Pivotal Cloud Foundry and is contributed to by many key members of the Cloud Foundry Foundation, such as Google, Pivotal, and VMware. It can support deployments across many different IaaS providers. Some of these providers are:

- VMware vSphere
- Google Compute Platform
- Amazon Web Services EC2
- Microsoft Azure
- OpenStack

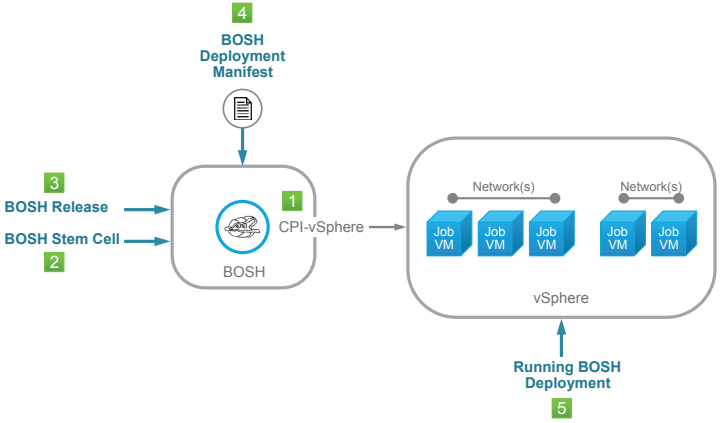

Figure 4: An overview of BOSH.

BOSH accomplishes deployments by creating several major abstraction objects that make it easy and repeatable to deploy complex systems. Referencing the figure above, these objects include:

1. CPI: The cloud provider interface, or CPI, is the executable library BOSH uses to interact with any given IaaS. One CPI is available for every BOSH-supported IaaS, and when you deploy the BOSH instances you can define which ones it will use. In the image above, a vSphere CPI is shown. It allows BOSH to perform all the required IaaS actions, such as creating a VM or instance, as well as various other instance, network, and storage primitives required to instantiate a deployment.

2. BOSH stemcell: A stemcell is a versioned base operating system image built for each CPI that BOSH supports. It is commonly based on Canonical's Ubuntu distribution, but is also available in RHEL and even Windows image ports. Typically, the stemcell is a hardened base OS image with a BOSH agent predeployed. BOSH will use this agent to install and manage the lifecycle of software on that VM for instance.
3. BOSH release: A BOSH release is a versioned tarball containing the complete source code and job definitions required to describe to BOSH how that release of software should be deployed on a VM or instance provisioned from a stemcell. An example is the Kubo release which includes all the packages and details required to allow BOSH deploy a fully functional Kubernetes cluster.
4. BOSH deployment manifest: BOSH needs to receive some declarative information to actually deploy something. This is provided by an operator via a manifest. A manifest defines one or more releases and stemcells to be used in a deployment and provides some key variables like IPstack info, instance count, and advanced configuration of the given release(s) you want to deploy. This manifest is typically written in a YAML format.
5. BOSH deployment: BOSH needs some declarative information before it can deploy anything. This is provided by an operator via a deployment manifest and a cloudconfig manifest. These manifests are typically written in a YAML format.

    - cloud-config manifest: This YAML is specific to an IaaS as defined by the properties made available in its CPI. It will provide definitions for things like networks, VM sizes, storage locations, and availability zone mappings. This manifest is global, which means that there can be only one instance per BOSH, and can be referred to by multiple deployment manifests.
    - deployment-manifest: This manifest refers to objects in the cloud-config and focuses on properties for the releases. The manifests define one or more releases and stemcells to be used in a deployment and provide some key variables like instance count and advanced configuration of the given release(s) to be deployed. This allows for deployment manifests to be portable across CPIs.

What Problems Does BOSH Solve?

BOSH lets release developers easily version, package, and deploy software in a reproducible manner. Operators can consume BOSH releases and be guaranteed that deployments are repeatable with predictable results across environments. To accomplish this, BOSH lets release developers focus on providing some key abilities when building a release:

Identifiability: An operator needs to be able to document the deployment of software and its versions. A BOSH release, by design, requires the developer to declare and package everything in the release. The release itself must also be versioned. This allows an operator to fully understand what is deployed as well as consistently upgrade or downgrade versions of software in a release.

Example: In Figure 2, an operator defining a deployment can refer to one or more versioned releases in a deployment manifest. This allows for identification of the software versions used. In the image above, BOSH has two versions of the Kubo release available: versions 0.0.5 and 0.0.6. The operator has defined the use of version 0.0.5 of the release in the deployment manifest, which will enforce the use of Kubernetes version 1.6.6 across the deployment called "mykubo-deployment."

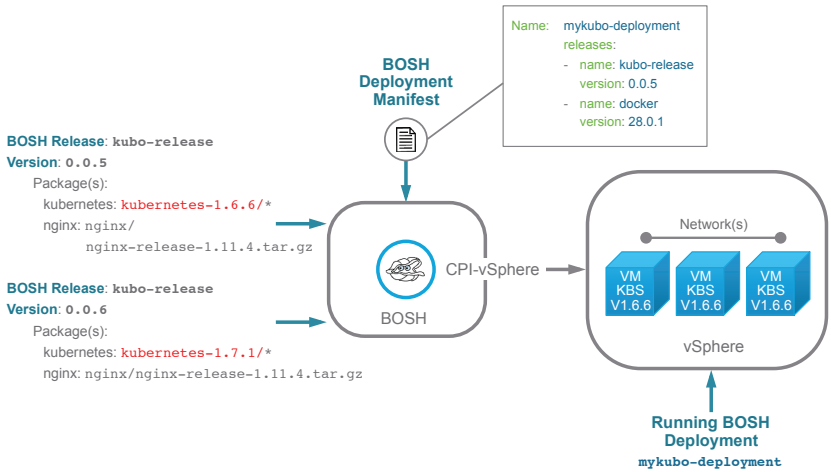

Figure 5: BOSH identifiability.

Reproducibility: Another key tenant in releasing software addressed by BOSH is reproducibility. To an operator, this means that software should be deployed in exactly the same way across multiple environments in order to guarantee operational stability.

Example: In the figure on the following page, a single manifest can deploy Kubernetes in a consistent way, providing the same functional deployment with the same releases across multiple environments. Those environments can even cross multiple IaaS providers by using the CPI abstraction. The simplified and partial deployment manifest in the image above is declaring which BOSH stemcell, BOSH Release, and config properties to use to deploy functionally identical Kubernetes clusters in two different environments.

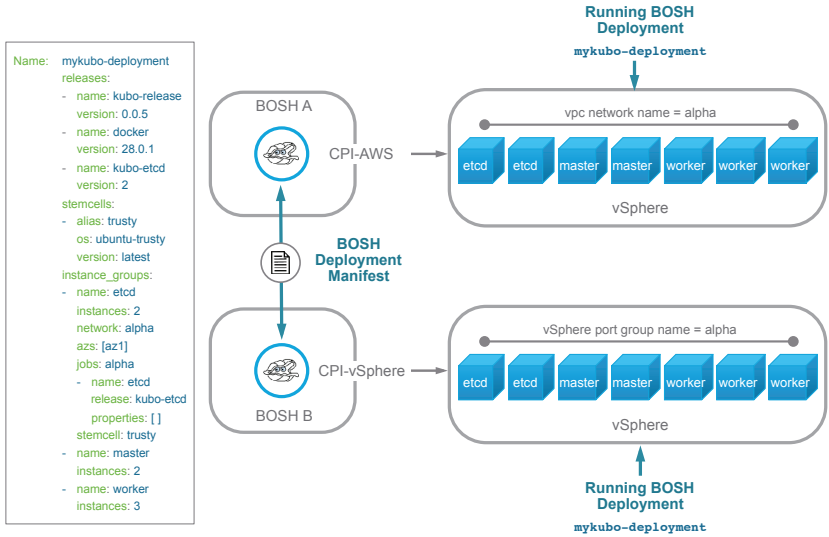

Figure 6: BOSH reproducibility.

Consistency: BOSH also enforces consistency in BOSH release development to ensure that virtually any software can be packaged, versioned, and deployed in a similar pattern. This also provides operational stability.

BOSH Use Cases and Benefits

BOSH's principal value lies in simplifying the deployment and day 2 lifecycle management of complex systems. It was primarily developed to deploy Cloud Foundry but has been extended by developers to deploy many other environments, both simple and complex. Systems to which BOSH can deploy can be found in two primary locations. The first is Pivotal Network, where Pivotal curates commercial BOSH releases of Pivotal Cloud Foundry as well as Pivotal services that are typically driven by Pivotal Operations Manager plus BOSH. The second location is BOSH.io, which hosts an OSS community repo of various systems that can be deployed.

A prime example of a BOSH use case is Kubernetes powered by BOSH, formerly known as Kubo.

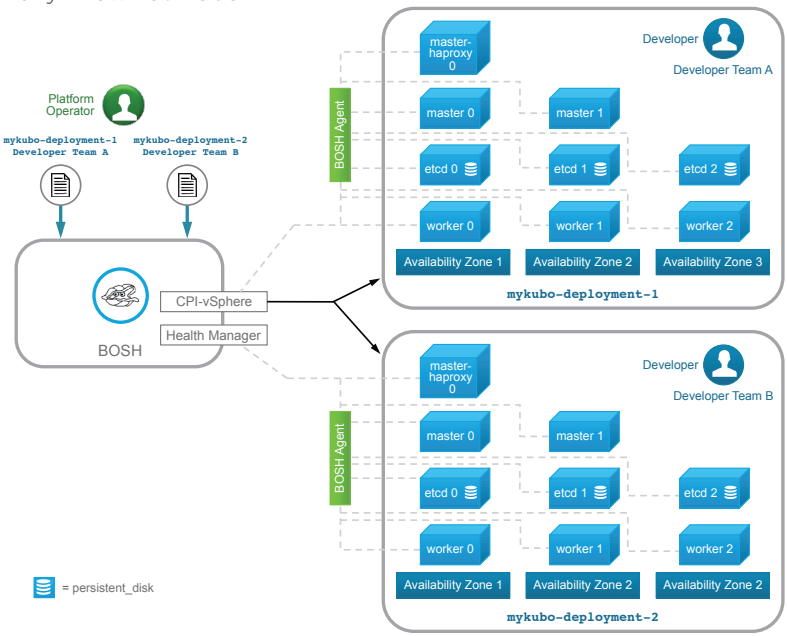

Figure 7: A use case with Kubernetes powered by BOSH.

Referencing Figure 8, we can see the key benefits that BOSH provides the operator.

1. Repeatability: In a cloud native development environment, the operator can generate two or more similar deployment manifests to deploy two or more unique but functionally identical Kubernetes deployments to meet the needs of multiple developer consumers.

2. Day 2 operations: BOSH lifecycle management makes it easy to keep all of the Kubernetes deployments healthy.

- Maintain health: Each VM or instance deployed by BOSH also deploys an agent that communicates health back to BOSH. If a Kubo node is unhealthy, BOSH will automatically try to repair and or rebuild the affected node. This improves uptime.
- Increase uptime: Each release job instance type can have multiple VMs or instances distributed across availability zones to ensure services provided are not affected by physical faults in a given availability zone. Availability zones are only supported on certain CPIs, such as the vSphere CPI where availability zones map to vCenter clusters.

- Patching: Because BOSH uses versioned releases, it is trivial for an operator to upgrade the Kubernetes Kubo release and apply it to all running deployments with little to no interruption of service. BOSH will update each deployment as well as maintain its state by: (1) detaching persistent disks, (2) rebuilding the affected VMs or instances, and then (3) re-attaching persistent disks.

BOSH Architecture

BOSH is typically deployed as a single VM or instance. That VM/instance has many components that perform vital roles in enabling BOSH to manage deployments at scale:

- NATS: Provides a message bus via which the various services of BOSH can interact.
- POSTGRESQL: BOSH writes all of its state into a database. Typically that database is internal to a single VM BOSH deployment and provided by Postgres. This can be modified, however, to use an external data source so that the BOSH VM can be rebuilt and reconnect to the database to reload its persistent state.
- BLOBSTORE: Each stemcell and release uploaded to BOSH is stored in a blobstore. Default deployments of BOSH use an internal store (webdav), but, like the Postgresql database, this can also be externalized.
- Director: The main API that the BOSH CLI will interface with to allow an operator to create and manage BOSH deployments.
- Health Monitor: BOSH requires that each VM it deploys have an agent that it can communicate with to assign and deploy jobs from BOSH releases that are defined in a deployment manifest. It will also maintain the health of each VM or instance it has deployed. The agent will report vitals back to BOSH and in cases where services in the VM are faulted, or the agent is unreachable, the Health Monitor can use plugins to restart services and even rebuild the VM or instance.
- CPI: The CPI is the IaaS-specific executable binary that BOSH uses to interact with the defined IaaS in its deployment YAML.
- UAA: Provides user access and authentication that allows BOSH to authenticate operators via SAML or LDAP backends.
- CREDHUB: Manages credentials like passwords, certificates, certificate authorities, SSH keys, RSA keys, and arbitrary values (strings and JSON blobs). BOSH will leverage credhub to create and store key credentials for deployments, like public certificates and keys.

- CLI: A final component, which is not shown in the architecture diagram, is the command-line interface. BOSH is deployed by using the BOSH CLI, passing the correct cmd line arguments, or storing those arguments as variable data within additional YAML files to define how BOSH itself will be deployed. This cookBook section outlines the steps required to deploy BOSH, and offers guidance for a basic Kubo deployment.

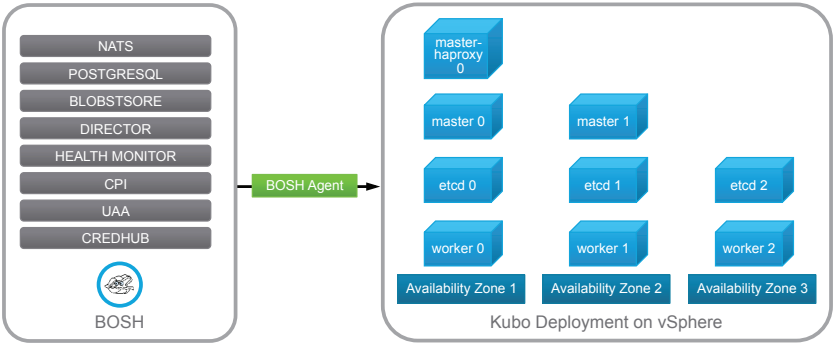

Figure 8: The architecture of BOSH.

For a step-by-step guide to deploying BOSH, see An Introduction to BOSH.

CFCN for Deploying and Operating Kubernetes

Cloud Foundry Container Runtime (CFCR), formerly known as Kubo, is an open source project that delivers the functionality of both Day 1 (deployment) and Day 2 (operations) for Kubernetes clusters. The initial genesis behind CFCR was to make deploying and running Kubernetes clusters across different environments more portable and operational.

Until now, there has been no reliable or convenient way to deliver a strong level of operational capability to a consumer who may want to run Kubernetes in production on their own on-premises and public clouds. To solve this problem, Google partnered with Pivotal (the leading contributor to BOSH) to build Cloud Foundry Container Runtime. CFCR was formerly known by the acronym Kubo, meaning Kubernetes on BOSH. BOSH is an open source tool for the deployment, release engineering, lifecycle management, and monitoring of distributed software systems. Google and Pivotal saw BOSH as a tool with the potential to facilitate production-grade Kubernetes operations.

Self-Care for the Kubernetes Control Plane

On its own, Kubernetes does a great job maintaining healthy running workloads. However, it's not so great at self-care of its control plane components like its API, controller manager, etc., or its core kubelet processes. BOSH provides health and monitoring capabilities to the complete Kubernetes control plane to keep not only app workloads healthy, but also Kubernetes itself healthy and running.

To accomplish this, CFCR is deployed a little differently than when deployed with tools like kubeadmin or kops. When BOSH deploys a Kubernetes cluster, each core component of the Kubernetes control plane is instantiated as a virtual machine (VM) instance. BOSH deploys an agent on each VM instance to monitor the health of the key Kubernetes control plane processes, as well as the overall health of each VM instance. BOSH will also dynamically repair and rebuild any VM that is unhealthy, no manual intervention required.

In addition to the health management of Kubernetes, CFCR deployments gain the added benefits of scaling, patching and upgrading Kubernetes clusters easily via a simple interaction with BOSH. Why is this so advantageous? Customers running Kubernetes will likely at some point need to upgrade and re-deploy, which is a taxing process. BOSH greatly simplifies this.

Alleviating the Operational Complexity of Kubernetes

Operating Kubernetes is difficult, generally speaking. CFCR was designed to address the complexity of Kubernetes deployment and make it easier to deploy, patch, upgrade, scale and operate. The BOSH approach to Kubernetes provides some nice advantages. One of those advantages is repeatability. BOSH can deploy Kubernetes across multiple IaaS providers, such as vSphere, Google Cloud Platform and Amazon Web Services. This is accomplished through the BOSH Cloud Provider Interface (CPI), which allows BOSH to create and manage VM instances, storage, and networking constructs across supported CPIs.

BOSH also utilizes a few additional abstractions, such as "releases" to package software, "stemcells" to define a secure VM image, and "manifests" to define how the releases get deployed across one or more VM instances based on the stemcells. A platform reliability engineer can use

these abstractions to make Kubernetes deployments easy and repeatable across any of the CPIs available to BOSH, thereby creating a great common operational model across any cloud.

Supplying Repeatable Automation with BOSH

Cloud Foundry Container Runtime also has a lot to offer when it comes to Day 2 operations, that is, operations that take place after the Kubernetes clusters have been deployed. Tasks like patching CVEs, upgrading Kubernetes or rotating key credentials can be pretty cumbersome. BOSH offers platform reliability engineers the ability to automate all of these tasks in a consistent and repeatable manner, driving down costs and the time to deliver software. Additionally, spinning up or decommissioning multiple Kubernetes clusters when they are no longer necessary can also be automated and logged in the BOSH database to provide a level of task auditing.

CFCR is a fundamental part of VMware Pivotal Container Service (PKS). The primary objectives of CFCR and PKS are to make Kubernetes and the operations of Kubernetes as a service simple and production ready. PKS is discussed in the next chapter.

Container Platforms and Services

The objective of this chapter is to help you understand container platforms from VMware, their underlying technology, and their business value so that you can make an informed decision about the best platform for your organization, its use cases, and its goals.

As you turn to container technology, an effective strategy for digital transformation includes matching the architecture and workloads of your applications to the right platform for the job. Different types of platforms are primed for different levels of container adoption, organizational requirements, and use cases.

High-Level Use Cases for Container Platforms

The following use cases coincide with the extent to which an enterprise has embraced container technology, defining a sort of container-adoption continuum that takes place after an initial stage of experimentation and evaluation:

- Establishing a developer sandbox or self-service agile infrastructure.
- Repackaging a legacy application in a container.
- Migrating, or replatforming, a traditional app to a container platform.
- Replatforming a legacy app and re-architecting, or refactoring, it by using microservices.
- Building cloud-native apps or developing on and for the cloud.

Maturity of Container Adoption

In the early stages of container adoption, organizations seek ready-to-go development tools and a service portal so that developers can self-service their needs with agile infrastructure. In the middle stage of the journey, organizations strive to accelerate software development by repackaging traditional applications in containers to simplify developer workflows and application maintenance. This kind of a lift-and-shift use case eventually leads to replatforming the repackaged application so that its deployment can be automated, orchestrated, and scaled on demand.

Cloud Natives

Then there are the cloud natives who live in and build for the cloud. They seek to build new applications by using architectural patterns like microservices and methodologies like the 12-factor app. When it makes sense to do so, they are also working to refactor legacy monolithic applications as cloud-native apps. Organizations at this stage are focused on automation, optimization, and rapid innovation.

Matching the Platform to the Project

Different platforms address different situations, and several factors come into play in analyzing the platform that's right for your stage of container adoption and your use cases:

- Identifying your target use cases or application types and matching them with the best-suited platform—that is, using the right tool for the job.
- Determining how much operational work is to be handled by DevOps or spread among developers and traditional IT teams—that is, having the right workers for the job.
- Deciding how much flexibility you want, including how you handle continuous integration, continuous delivery, and continuous deployment—that is, finding the right balance between prescription and complexity.

Prescription and Complexity

The more prescriptive the platform is, the more it hides the complexity of the platform from developers. A prescriptive platform prescribes a scheduler, a runtime engine, integration with the underlying infrastructure, continuous delivery, and other aspects of the platform. (A prescriptive platform is also referred to as an opinionated platform.)

For example, a prescriptive platform includes its own scheduler for containers and specifies how to use it to run containerized applications. The main advantage of a perscriptive platform is that it places the platform's complexity in a layer of abstraction—all developers have to do is write their code and generate an application artifact, and the platform handles the rest. The disadvantage is that you have fewer options and less flexibility in how you delivery and deploy your app. A prescriptive platform also imposes methods of using containers on you; as a result, you might be unable to manage containers with standard APIs, such as the Docker API or the Kubernetes API.

FACTORS IN SELECTING A CONTAINER PLATFORM

The platform that you select will depend on your unique situation and goals. Here are some factors to consider:

- Use cases
- Application types and their workloads
- Software development methods and processes
- Operations
- Security and compliance
- Networking
- People and their skill sets
- Maturity of your organization's container adoption
- Maximizing flexibility vs. minimizing complexity
- Business objectives

Closing the Container Platform Confidence Gap

In a recent report titled "Closing the Digital Transformation Confidence Gap in 2017," The Hackett Group surveyed executives from more than 180 large companies. The report found a wide confidence gap "between the high expectations for digital transformation's business impact and the low perception of the business's capability to execute digital transformation." The Hackett group says that the findings demonstrate the need for IT to invest in the necessary tools and to adopt rapid application development techniques, such as agile processes.[11]

Although containers themselves are not new, barriers have hindered their use for building and deploying enterprise applications. Until fairly recently, containers lacked the tooling and ecosystem for enterprise-grade deployment, management, operations, security, and scalability. In addition, the requirements of IT administrators often went unfulfilled: Infrastructure for running containers has neglected networking, storage, monitoring, logging, backup, disaster recovery, maintenance, and high availability.

[11]Despite High Expectations for Digital Transformation Led by Cloud, Analytics, Robotic Process Automation, Cognitive & Mobile, IT & Other Business Services Areas See Low Capability to Execute, The Hackett Group, March 16, 2017. A version of the research is available for download, following registration, at http://www.thehackettgroup.com/research/2017/social-media/key17it/.

VMware vSphere Integrated Containers and VMware Pivotal Container Service changes all that. These cloud-native solutions from VMware help you quickly and cost-effectively put containers into production, improving your ability to carry out digital transformation.

Running containers on VMs also adds a beneficial level of security to containerized applications, especially in the context of the third tenet of cloud-native applications—microservices. According to a Docker white paper on security, "Deploying Docker containers in conjunction with VMs allows an entire group of services to be isolated from each other and then grouped inside of a virtual machine host."[12]

Deploying containers with VMs encases an application with two layers of isolation, an approach that is well-suited to cloud-style environments with multitenancy and multiple workloads. "Docker containers pair well with virtualization technologies by protecting the virtual machine itself and providing defense in-depth for the host," the Docker security white paper says. Container security is discussed further in a later section.

vSphere Integrated Containers

VMware vSphere Integrated Containers, a comprehensive container solution built on VMware vSphere, enables you to run modern and traditional workloads in production on their existing software-defined data center (SDDC) infrastructure with enterprise-grade networking, storage, security, performance, and visibility.

vSphere Integrated Containers offers the quickest and easiest way for vSphere users to start using containers today without additional capital or labor investment. Its tight integration with the entire VMware SDDC environment, as well as its support for leading container technologies like Docker, makes it a great solution for a seamless transition to container adoption. You can tap the benefits of containers for enhanced developer productivity, business agility, and fast time-to-market.

The following sections examine the architecture and capabilities of vSphere Integrated Containers when deployed as an integrated system with VMware vSphere, NSX, vSAN, and several external systems, including Microsoft Active Directory and Docker Hub.

Before diving into the details of the system architecture, here's a brief review of the system's design objectives. vSphere Integrated Containers addresses the following commonly occurring objectives:

---

[12] Introduction to Container Security, Docker white paper, Docker.com.

- Enable a universal platform for transitioning to modern development practices.
- Enable the infrastructure to support the coexistence of both traditional and modern application designs on common, existing hardware and software.
- Improve developer agility, shorten time to market, and maximize application resiliency
- Developers need an environment where they can build, test, and run their applications using native container tools with minimal involvement from IT.
- Support a standard framework for orchestrating the deployment of cloud native applications and automating management of application availability in operation.
- Provide integration with the enterprise-grade capabilities of VMware infrastructure.
- Provide security and availability of application when running in production.
- Increase visibility into container deployments using standard VMware tools for better operability.
- Streamline development team access to tools and infrastructure resources.
- Eliminate extensive approval processes for acquisition and manual provisioning of infrastructure, which frequently results in developers pursuing alternative paths of less resistance such as rogue IT or public offerings.

Architecture

vSphere Integrated Containers is a product designed to tightly integrate container workflow, lifecycle and provisioning with the vSphere SDDC. It provides a container management portal, an enterprise-class registry, and a container runtime for vSphere fully integrated into a commercial distribution.

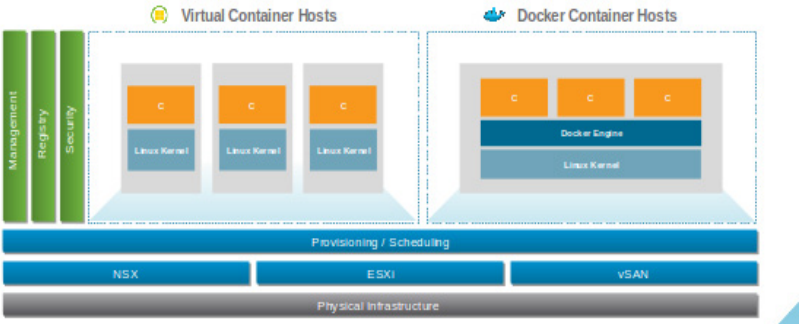

Figure 9: The architecture of vSphere Integrated Containers.

With these capabilities, vSphere Integrated Containers enables VMware customers to deliver a production-ready container solution to their developers and DevOps teams. By leveraging their existing SDDC, customers can run container-based applications alongside existing virtual machine based workloads in production without having to build out a separate, specialized container infrastructure stack.

As an added benefit for customers and partners, vSphere Integrated Containers is modular. So, for example, if your organization already has a container registry in production, you can use that registry with vSphere Integrated Containers Engine and vSphere Integrated Containers Management Portal.

Components

vSphere Integrated Containers is built on these major open source products:

1. vSphere Integrated Containers Engine is a container runtime for vSphere that enables the provisioning and management of applications packaged as Docker images into vSphere clusters. With the vSphere Integrated Containers Engine Developers can deploy container images alongside traditional workloads on vSphere clusters. The vSphere Integrated Containers engine gives developers the agility and speed they need, while allowing operations to reuse the tools, processes and people they've already invested in.
2. Harbor is an enterprise-class private container registry that stores and distributes container images. It extends the Docker Distribution

open source project by adding the functionalities that enterprises require, such as security, auditing and identity management.

3. Admiral is a container management portal. It provides a GUI for DevOps teams to provision and manage containers, and includes the ability to obtain statistics and information about container instances. It provides both Docker compose and a proprietary application definition through templating to combine different containers into an application. It also supports containers scaling in and out. Advanced capabilities, such as approval workflows, are available when integrated with vRealize Automation.
4. Photon OS is a minimal Linux container host, optimized to run on VMware platforms. It is used throughout vSphere Integrated Containers wherever a Linux guest kernel is required.

The core SDDC infrastructure subsystems, vSphere, NSX, and vSAN complement vSphere Integrated Containers by extending trusted capabilities such as:

- Distributed Resoource Scheduling (DRS)
- vMotion
- High Availability (HA)
- Secure isolation, micro-segmentation, and RBAC
- SSO via PSC with extension to external identity sources such as Active Directory/LDAP
- Granular monitoring and logging visibility via vCenter, vRealize Operations, and VRNI
- vSAN, iSCSI, NFS shared storage
- Direct deployment to Distributed vSwitch and NSX Logical Switches, and integration with NSX virtual network infrastructure components
- Unified, full-stack monitoring and logging visibility

Deployment Options

vSphere Integrated Containers supports multiple ways to deploy and run containers. Its deep integration with an existing VMware SDDC environment provides the best of both worlds for your developers and IT staff, while supporting a variety of container use cases for the modern enterprise.

Small businesses to large enterprises can leverage the capabilities of vSphere Integrated Containers as it deploys to vSphere infrastructures of various sizes. A container deployment model can, for example, overlay typical management and control plane components along with the two container deployment model options over two vSphere clusters of ESXi hosts. Traditional VMs could also exist on a common cluster with the containers.

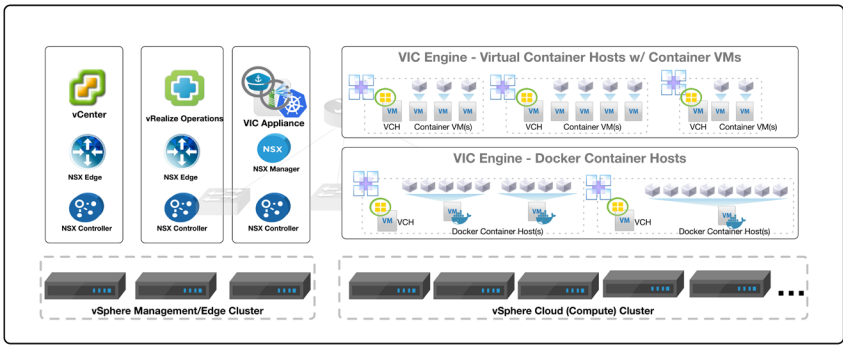

Figure 10: Container deployment models.

Virtual Container Hosts

vSphere Integrated Containers leverages the native constructs of vSphere for provisioning container-based applications. IT admins can deliver a production-ready container solution to their developers and app teams without having to build out a separate, specialized container infrastructure stack. By deploying each container image as a vSphere Virtual Machine (VM), vSphere Integrated Containers allows these container workloads to leverage critical vSphere application security, isolation, availability and performance features – VMware HA, vMotion and Distributed Resource Scheduler. vSphere Integrated Containers provides these features while still presenting a Docker API to developers of container based applications to consume.

Docker Container Hosts

vSphere Integrated Containers also supports running native Docker container hosts on vSphere. It allows developers to self-provision Docker container hosts for use as a development sandbox, a build server, or a swarm cluster. Now you can treat a Docker host as ephemerally as a container.

Container Runtime

The vSphere Integrated Containers Engine is a container runtime for vSphere. It enables the provisioning and management of VMs into vSphere clusters using the Docker binary image format. It enables vSphere admins to pre-allocate certain amounts of compute, networking and storage and provide that to developers as a self-service portal exposing a familiar Docker-compatible API. It allows developers who are familiar with Docker to develop in containers and deploy them alongside traditional VM-based workloads on vSphere clusters. VMs provisioned using vSphere Integrated Containers take advantage of many of the benefits of vSphere including DRS, clustering, vMotion, HA, distributed port groups and shared storage.

Using the native constructs of vSphere, IT admins can deliver a production-ready container solution to their developers and app teams without having to build out a separate, specialized container infrastructure stack.

vSphere Integrated Containers engine brings many of the value propositions of containers and container images directly to vSphere infrastructure. vSphere Integrated Containers engine turns container images into objects that look just like containers when viewed from a developer's perspective and look just like a virtual machine when viewed from an operator's perspective. Since vSphere Integrated Containers expose the Docker API, it is easy to integrate with developer tools, scripts and processes. And since they behave just like virtual machines, vCenter, NSX, vRealize Operations, vSAN, vMotion and other familiar technologies are just as relevant and valuable for container workloads.

The VMs created by vSphere Integrated Containers engine have all of the characteristics of software containers:

- Ephemeral storage layer with optionally attached persistent "volumes"
- Custom Linux guest designed to be "just a kernel" needs "images" to be functional
- Automatically configured to various network topologies

"ContainerVMs" are provisioned into a "Virtual Container Host" which represents a clustered pool of resource, a single-tenant container namespace and an API endpoint. A VCH is not a literal host, rather it is akin to a vSphere resource pool in that it transparently provides clustering, scheduling, vMotion and HA to containers running in it. A VCH is represented in vSphere as a resource pool construct.

All of the basic capabilities for creating VMs with these characteristics along with the ability to configure the necessary networking, compute and storage to support them is encapsulated in the “Port Layer” (4.4) service. The Port Layer also adds capabilities to listen for events and interact with the containers.

The scope of vSphere Integrated Containers engine is limited to being an endpoint that runs production container workloads. There is no native support for building images directly on the engine. However, vSphere Integrated Containers can be used by developers to provision native Docker Host VMs to be used for development and build in that manner. The fact that a native Docker Host is controlled from the exact same client as a VCH makes the experience relatively seamless.

vSphere Integrated Containers is optimized for container uptime and availability. Upgrading vSphere Integrated Containers momentarily impacts endpoint availability, but not the containers. Modifying per-tenant compute limits is completely transparent. Upgrading ESXi is also transparently handled with vMotion.

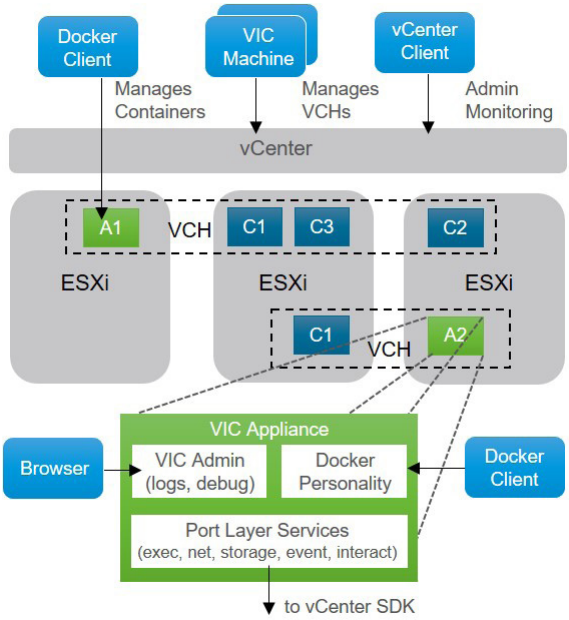

Figure 11: A conceptual model of vSphere Integrated Containers.

If you consider a Venn diagram of "What vSphere Does" in one circle and "What Docker Does" in another, the intersection is not insignificant. vSphere Integrated Containers takes as much of vSphere as possible and layers on whatever Docker capabilities are missing. The following sections discuss the key concepts and components that make this possible.

The Virtual Container Host

A Virtual Container Host, or VCH, is the virtual functional equivalent of a Linux VM running Docker. From a Docker client point of view, the Virtual Container Host looks very similar to a native Docker host. Hence, there are differences between a native Docker host and a Virtual Container Host (VCH), and between a Linux container and a container VM. Some of those differences are intentional design constraints, such as there being no such thing as a "privileged" container in vSphere Integrated Containers. Some are because of a lack of functional completeness, some are outside of the existing scope of the product, such as native support for docker build.

To use the standard command line tools, simply set DOCKER_HOST to point to your virtual Docker host. Commands like docker run, docker volume and docker net will work similarly as they do with a standard Docker host. However, in the contest of vSphere Integrated Containers, the docker run command creates and provisions a container VM. Docker network commands allow container workloads to be connected to vSphere networks. Docker volume commands allow for the creation and lifecycle management of disks on vSphere storage.

A VCH is deployed as a resource pool in a vCenter Server cluster. The resource pool provides a useful visual parent-child relationship in the vSphere Client so that you can easily identify the containerVMs that are provisioned into a VCH. The first VM deployed inside the resource pool provides a secure Docker compatible API endpoint and other services for the VCH to run.

A VCH is functionally distinct from a traditional container host in the following ways:

- It naturally encapsulates clustering and scheduling by provisioning to vSphere targets
- The resource constraints are dynamically configurable with no impact to the containers
- The containers don't share a kernel. They could in theory run different kernels
- There is no local image cache. This is kept on a vSphere datastore

The VCH includes all containerVMs instantiated via docker run command and provides networking, storage, clustering, scheduling, vMotion, host evacuation and HA capabilities.

A single ESXi host can have multiple VCHs, each of which with different resources and different users. Similarly, a single VCH can expose the entire capacity of a vSphere cluster of ESXi hosts. It all depends on your own use case and requirements.

The lifecycle of the VCH is managed by a binary called vic-machine, which installs, upgrades, deletes and enables debugging for the VCH.

The Virtual Container Host API End-point

There is a 1:1 relationship between a VCH and a VCH API end-point. It is built off the same Photon OS Linux kernel as the containerVMs and is stateless. It has the following functions:

- Run the Core services, Docker personality and image resolution services
- Provide a secure remote Docker API
- Port mapping and routing - When a container port is mapped to a host port, the Virtual Container Host is responsible for listening on that port and routing traffic to the corresponding container VM
- Manage the lifecycle of containerVMs, image store, volume store and container state
- Provide logging and monitoring of its own services and of its containers

The VCH VM is completely stateless. The state is either on datastores, in VMX guestinfo or in vCenter itself. This makes upgrade a simple process of power down, swap ISO, power up, rediscover.

Security of the VCH VM is an important consideration. It is isolated from the containers, isolated from the vSphere management network and there is no ability to get a remote shell into it without explicit configuration.

The vic-machine Utility

The vic-machine utility is a binary built for Windows, Linux and Mac OSX that manages the lifecycle of VCHs. The vic-machine has been designed to be used by vSphere admins. It takes pre-existing compute, network, storage and a vSphere user as input and creates a VCH as output. It has the following additional functions:

- Creates certificates for Docker client TLS authentication
- Checks that prerequisites have been met on the cluster (firewall, licenses, etc.)
- Assists in configuring ESXi host firewalls
- Configures a running VCH for debugging
- Lists, reconfigure, upgrades/downgrades and deletes VCHs.

The vSphere Integrated Containers machine requires a vSphere admin user for the installation, but takes a separate "proxy" user for client operations. Operations from each VCH can then be audited under the name of the proxy user.

The Docker Personality

vSphere Integrated Containers engine supports version 1.25 of the Docker API, however not all commands and options are implemented. This is because the main target use case for vSphere Integrated Containers Engine is to run applications vs. build applications. The Docker client will report "not implemented" for anything the engine doesn't support.

The ContainerVM, OS, and Tether

As already stated, a container VM is a VM with all the characteristics of a container. To be clear, the provisioned VM does not contain any OS container abstraction. The VM boots from an ISO containing the Photon Linux kernel and is configured with container images mounted as a disk. Container image layers are represented as a read-only VMDK snapshot hierarchy on VMFS. At the top of this hierarchy is a read-write snapshot that stores ephemeral state. Container volumes are formatted VMDKs attached as disks and indexed on VMFS. Networks are distributed port groups attached as vNICs.

When the VM powers on, it boots from the ISO, chroots into the container filesystem on the attached disk, sets up any internal state such as environment variables and then starts the container process.

Interaction with a running container VM is managed by a "Tether" init process that runs as PID 1 inside the container VM. It is responsible for intermediating interaction (streaming stderr, tty etc) between the container and the client. It also manages the lifecycle of container processes and publishes the exit code when it terminates. The tether communicates with the vSphere Integrated Containers appliance via a virtual serial port concentrator.

Docker Container Host

vSphere Integrated Containers Engine also supports running native Docker container hosts on vSphere. It allows developers to self-provision Docker container hosts, and then use native Docker commands to build and run applications inside those Docker hosts.

Virtual Container Hosts vs. Docker Container Hosts

The vSphere Integrated Containers Engine enables two methods for deploying containers: Virtual Container Hosts and Docker Container Hosts (DCH). The following table summarizes the differences between the two deployment options:

FUNCTION	VIRTUAL CONTAINER HOST	DOCKER CONTAINER HOST
Docker Client Tools	Partial Compatibility (optimized for Run stage)	Full Compatibility (Optimized for Build stage)
Provisioning Process	VI Admin provisions the VCH; Developers provision containers as VMs Speed of Container Deployment Around 10 secs Around 2 secs	VI Admin provisions the VCH; Developers provision Docker Container Hosts and containers inside the DCH
Runtime Performance	Very Fast	Fast
Governance	Micro segmentation between individual containers as VMs via NSX	Does not provide network security between containers
Resource consumption	Memory is consumed for the lifecycle of the container workload	Memory is consumed for the lifecycle of the DCH

You can look at the Docker Container Host as a container VM that delivers a particular use case. Instead of instantiating, as a container VM, a Docker image that represents an application, you are instantiating, as a container VM, a Docker image that represents a Docker host.

Management Portal and Registry

Cloud admins and developers can manage and provision container-based applications through the vSphere Integrated Containers management portal. Integrated with VMware Identity Access Management, customers

are able to provide local and LDAP-based authentication and authorization to their teams and project-level content trust and notary services for container images in their private registries. Manual and automated container image vulnerability scanning is also included to avoid running images with known vulnerabilities in your data centers.

The management portal provides a UI for DevOps teams to provision and manage containers, including the ability to obtain statistics and information about container instances. Cloud administrators can manage container hosts and apply governance to their usage, including capacity quotas. Administrators can manage identity sources (local and external), users and groups, roles, and other credentials.

The Management portal also provides the following

- Rule-based resource management, allowing DevOps administrators to set deployment preferences for container placement
- Live state updates that provide a live view of the container system
- Multi-container template management, that enables logical multi-container application deployments
- Basic scale in and scale out of number of containers in a multi-container app template
- Enterprise-class private container registry

The container registry stores and distributes container images. Through the Management Portal DevOps administrators can organize image repositories in projects, and to set up role-based access control to those projects to define which users can access which repositories. The registry also provides rule-based replication of images between registries, implements Docker Content Trust, and provides detailed logging for project and user auditing. It extends the Docker Distribution open source project by adding the functionalities that an enterprise requires, such as security, identity and management. In addition to user authentication and RBAC, vSphere Integrated Containers Registry enables other security controls, namely:

- Content trust: Image signing and verification to content and version for ensuring security and auditability when running containers in production
- Vulnerability scanning: Traditionally, operating systems have been managed (specifically, patched on a regular basis) by ops personnel with developers providing only the application-level executables. However, containers often use base images like Ubuntu and CentOS from DockerHub, which contain hundreds of features, each of which is susceptible to vulnerabilities. Since container

images are essentially opaque to ops personnel, having vulnerability scanning helps IT Ops to prevent exploitation of known vulnerabilities when deploying these applications to production

Interacting with vSphere

This section explains how vSphere Integrated Containers interacts with vSphere.

Control Plane

The VCH VM acts as a proxy between the Docker client and the vSphere SDK and all of the control plane operations of a VCH are initiated by the vSphere user associated with it. As previously mentioned, the control plane is extended into containerVMs via the Tether process. The majority of control plane operations are VM creation, reconfiguration and deletion.

Given the multi-tenant nature of vSphere, it should be expected that there are multiple VCHs running concurrently in a vSphere cluster and multiple Docker clients connected to each VCH. Most control plane operations that result in a container state transition are synchronous API calls. The VCH API end-point handles blocking and queuing of concurrent Docker clients. In terms of vSphere sessions, the VCH appliance keeps a single session open and multiple connections are made over that session for control plane operations. The vicadmin web UI opens an additional session as the user needs to authenticate with it using vSphere credentials.

Compute

Compute is limited at the container level by number of CPUs and memory. This can be set from the Docker client. Compute is limited at the macro level by memory and CPU limits either on the VCH or a Resource Pool it's deployed into. One difference between vSphere Integrated Containers and regular Docker is that there's no such thing as an "unlimited" container. A VM necessarily has to have limits. As such, there is a default container VM configuration associated with a VCH.

Networking

vSphere Integrated Containers engine uses pre-configured vSphere port groups for its networking: either regular port groups, distributed port groups or logical switches created by NSX. It is designed to allow different types of traffic to be isolated on distinct networks. It is also designed

to allow vSphere networks to be directly exposed to the Docker client for private container traffic. It is the use of distributed port groups that allows for containerVMs to be provisioned across multiple hosts and vMotioned.

Networks must be created ahead of VCH creation and are specified as input to vic-machine, vSphere networks are exposed as "container networks" in Docker. vSphere Integrated Containers does not attempt to create or configure networks in vSphere. It is possible to specify different networks for the following:

- Expose Docker API traffic from Client to VCH
- Container traffic bridged to the VCH appliance
- Public network for image downloading and uploading
- vSphere management traffic
- Exposure of vSphere networks directly to containers

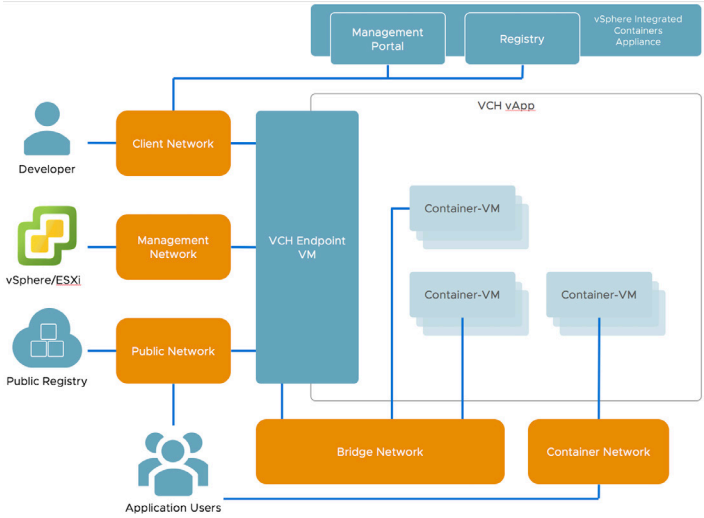

Figure 12: Networking in vSphere Integrated Containers.

All of these networks default to DHCP, but it is possible to specify IP ranges and gateways if required. Likewise, the appliance itself can be given a static IP address or use DHCP.

Networks created via the Docker client currently use IPAM segregation rather than full micro-segmentation.

Note that there is no special in-guest networking integration required for vSphere Integrated Containers containers. The container process talks through interfaces directly corresponding to vNICs.

Storage

As previously stated, vSphere Integrated Containers uses VMDKs on VMFS for all container storage. It supports any VMFS datastore, including vSAN, iSCSI, or local datastores. And it provides shared storage between containers by using an NFS volume driver.

As input to vSphere Integrated Containers machine, a user can specify different datastores for different types of container state. That's container ephemeral state, read-only image state and volume state. It is to be expected that different characteristics will be desirable for different kinds of state - for example, a customer is likely to want to back up their volumes, but not their container state.

When images are pulled from a Docker registry, they are extracted onto VMDK snapshots and indexed on a local datastore. Multiple containerVMs can share the same base images because they are immutable and mounted read-only.

Other vSphere Features

Here's a listing of how vSphere Integrated Containers interacts with other aspects of vSphere:

- vMotion is supported.
- Cross Cluster vMotion is unsupported.
- Distributed Resource Scheduller (DRS) is supported.
- High Availability (HA) is supported.
- Fault Tolerance (FT) is unsupported.
- VMware vSAN is supported.
- Virtual Volumes (VVOL) is unsupported.
- Snapshot is unsupported.
- Storage DRS: You cannot point to a storage DRS cluster but can consume individual datastores within it.

Benefits of the Container VM Model

A container VM is strongly isolated by design and benefits from vSphere enterprise features such as High Availability and vMotion. It is ideally suited to long-running containers or services with the following requirements:

- Strong isolation - a container VM has its own kernel and has no access to a shared filesystem or control plane
- High throughput - a container VM has its own guest buffer cache and can connect directly to a virtual network
- High availability - a container VM can be configured so it can run independent of the availability of the VCH and can benefit from vSphere HA and vMotion
- Persistent data - a container VM can persist its data to a volume disk that can be backed up completely independent of the VM

This means that it is not possible to deploy a container with access to the control plane. It is also impossible to mount parts of the host's filesystem as shared read-write volumes into the container.

vSphere Integrated Containers containers are slower to start and use more memory resource than Linux containers. ContainerVMs have to be placed, configured and booted. However, in terms of runtime performance, vSphere Integrated Containers containers show improved throughput. Improved throughput is due to not having the additional layer of OS virtualization in the guest.

Running containers in containerVMs makes a lot of sense for long-running services. If the service fails, it should have no impact on any other services. Examples of a long-running service are a database, web server, key-value store etc.

A container VM is less well suited to containers that are transactional and have a very short lifespan, such as running a unit test. This is because the cost to boot the VM is high relative to the time spent running the test. A container VM however is very well suited to longer-running transactional workloads, such as builds. This is because vSphere resource is only consumed for the period of execution and is immediately freed up after. This can lead to a much more efficient use of virtual infrastructure than slave VMs that are up all the time waiting for jobs.

A container VM is also less well suited to containers that need to be weakly isolated by design, for example a logging or monitoring container

that need access to the other processes in an application. This is also true of very small containers that together make up a single service or unit of scale. For this purpose, the VM is the ideal isolation domain for the service as a whole and the containers can be deployed inside the VM as software containers using a regular container engine.

vSphere Integrated Containers is a great way to manage regular container hosts, because the container VM abstraction allows you to treat them just as ephemerally as containers.

When deploying applications into production, it's important to consider where the isolation boundaries should lie for your particular container, service or application. A VM is a natural isolation and failure domain and works well as a unit of scale.

Benefits of DCH over a Docker engine deployed in a VM

Deploying infrastructure to support application development is often cumbersome, error-prone, and time-consuming. As developers rush to build new apps, IT teams waste time with manual configuration, provisioning, and scripting. To improve productivity, they need, at the very least, to streamline the way they roll out and manage infrastructure for developers to use.

Modern developers need an environment where they can build and test their apps using native container technology with minimal involvement from IT. Today, they use their laptops or a VM with a Docker engine in it as the main tools to build containerized applications. However, trying to build an application that goes beyond a simple demo on a laptop or desktop can hit performance and memory constraints. And having developers requesting a VM with the Docker engine in it from IT is time consuming because all the burden of configuration management and network configuration is left to the IT team.

The key solution is providing developers with a secure sandbox so they can serve their own development needs by letting them create native Docker container hosts on demand on vSphere using the Docker CLI they love. By using the Docker container hosts (DCH) feature, developers can deploy Docker container hosts within a vSphere resource pool without having to file a ticket with IT. The create, run, stop, and delete operations are all handled using the native Docker CLI/API.

For example, developers can use DCH to integrate VIC into their CI/CD pipeline and use products like Jenkins to build applications on DCH and then push them to production using VCH. This allows build and test jobs to use vSphere infrastructure as completely ephemeral compute.

The DCH gives developers the Docker tools they need to build modern applications or repackage existing ones and IT teams governance and control over the infrastructure. vSphere administrators provision compute, networking, and storage resources and provide them to developers as a self-service portal that exposes the familiar Docker compatible API.

The DCH provisioned using vSphere Integrated Containers has also a much-reduced attack surface because no extra services besides the Docker daemon are installed and only access to the remote Docker API is provided.

Moreover, the DCH can take advantage of many of the benefits of vSphere, including Distributed Resource Scheduler, clustering, VMware vSphere vMotion®, VMware vSphere High Availability (HA), distributed port groups, and shared storage making it a very robust development infrastructure.

Developers and IT teams need not worry about patching, security, isolation, of the Docker hosts. Those functions are completely automated by how DCHs are deployed as part of VIC.

The outcome is a win-win situation for both developers and administrators: The vSphere administrator gets visibility into and control over the virtual infrastructure, while developers can self-provision Docker container hosts and work with them by using a Docker client.

Security and Isolation

Security and isolation are among the biggest differentiators of vSphere Integrated Containers. Here is a high-level list of security features:

- Docker client authenticates with VCH using a certificate by default
- Network isolation is achieved through multiple port groups
- vSphere Integrated Containers appliance is locked down by default
- VM isolation, every container is fully isolated from the host and from other containers
- Containers are completely isolated from each other and the ESXi hosts
- vSphere Integrated Containers supports authentication with a secure registry
- vSphere Integrated Containers supports strong identity and access management (IAM) with LDAP and Active Directory services

- vSphere Integrated Containers enables administrators to control access at the project level, ensuring granular security across teams
- vSphere credentials persisted in ExtraConfig so they are not visible to the appliance guest
- Enterprise private container registry: With Harbor, vSphere Integrated Containers offers an enterprise private container registry with advanced security features such as identity management, LDAP integration, role-based access control, and trusted content, all of which help ensure security for container images. With the private registry, you can furnish project-level content trust and notary services to container images. Vulnerability scanning helps prevent vulnerable container images from running in your data center.

Installation and Configuration

You install vSphere Integrated Containers by deploying an OVA appliance. The OVA appliance provides access to all of the vSphere Integrated Containers components.

The installation process involves several steps.

1. Download the OVA from VMware web site.
2. Deploy the OVA, providing configuration information for vSphere Integrated Containers. The OVA deploys an appliance VM that runs vSphere Integrated Containers Management Portal and Registry; Makes the vSphere Integrated Containers Engine binaries available for download; and hosts the vSphere Client plug-in packages for vCenter Server.
3. Run the scripts to install the vSphere Client plug-ins on vCenter Server.
4. Run the command line utility, vic-machine, to deploy and manage virtual container hosts

Summary

VMware vSphere Integrated Containers is a comprehensive container solution built on the industry-leading virtualization platform, VMware vSphere. It enables customers to run both modern and traditional workloads in production on their existing SDDC infrastructure today with enterprise-grade networking, storage, security, performance and visibility. It offers the quickest and easiest way for vSphere customers to start using containers today without additional capital or labor investment.

VMware Pivotal Container Service

VMware Pivotal Container Service (PKS) provides a production-grade Kubernetes-based container solution equipped with advanced networking, a private container registry, and full lifecycle management. The solution radically simplifies the deployment and operation of Kubernetes clusters so you can run and manage containers at scale on VMware vSphere or in public clouds.

With hardened production-grade capabilities, PKS can manage your container deployment from the application layer all the way to the infrastructure layer. Critical production capabilities include high availability, auto-scaling, health-checks and self-healing of underlying VMs, and rolling upgrades for Kubernetes clusters. Constant compatibility Google Kubernetes Engine (GKE) ensures that developers get the latest stable Kubernetes release, features, and tools.

PKS integrates with VMware NSX-T for advanced container networking, including micro-segmentation, ingress controller, load balancing, and security policy.

An integrated private registry secures container images with vulnerability scanning, image signing, and auditing. In addition, with the VMware SDDC portfolio, enterprises can also use persistent volumes and integrate with operational tooling such as monitoring, logging, and analytics.

PKS exposes Kubernetes in its native form without adding any layers of abstraction or proprietary extensions, which lets developers use the native Kubernetes CLI that they are most familiar with. PKS can be deployed and operationalized by using Pivotal Operations Manager, which allows a common operating model to deploy PKS across multiple IaaS abstractions like vSphere and Google Cloud Platform.

PKS is certified by the Cloud Native Computing Foundation (CNCF) through its Kubernetes Software Conformance Certification program. This certification lets you run applications with the confidence that the Kubernetes deployment has passed CNCF test suites and is compliant with the community's specification. As more organizations adopt Kubernetes, a certified Kubernetes product like PKS ensures portability, interoperability and consistency between different environments.

KEY BENEFITS OF VMWARE PIVOTAL CONTAINER SERVICE

- Eliminate lengthy deployment and management process with on-demand provisioning, scaling, patching and updating of Kubernetes clusters through a simple CLI or API.
- Access the latest stable Kubernetes release and gain constant compatibility with Google Kubernetes Engine (GKE).
- Provide high availability for Kubernetes components (master, worker, etcd nodes) with rolling upgrades, health-checks, and auto-healing of underlying virtual infrastructure
- Simplify container networking and increase security with Vmware NSX, providing high availability, automated provisioning, micro-segmentation, ingress controller, load balancing, and security policy.
- Deploy Kubernetes clusters for both stateless and stateful applications.
- Secure application deployments with an integrated enterprise container registry with vulnerability scanning, image signing, and auditing.
- Improve operational efficiency with monitoring, logging, and analytics.

Architecture

PKS builds on Kubernetes, BOSH, VMware NSX-T, and Project Harbor to form a highly available, production-grade container service. With built-in intelligence and integration, PKS ties all these open source and commercial modules together, delivering a simple-to-use solution with an efficient Kubernetes deployment and management experience.

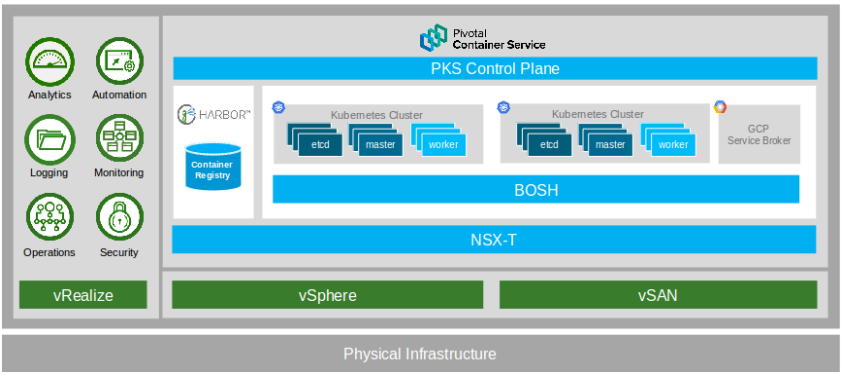

Figure 13: The architecture of VMware Pivotal Container Service.

PKS Control Plane

A key component of PKS, the control plane is the self-service interface responsible for the on-demand deployment and lifecycle management of Kubernetes clusters. It provides an API interface for self-service consumption of Kubernetes clusters. The API submits requests to BOSH, which automates the creation, update, and deletion of Kubernetes clusters.

Operations and Automation with BOSH

BOSH is an open source tool for release engineering that simplifies the deployment and lifecycle management of large distributed systems. With BOSH, developers can easily version, package, and deploy software in a consistent and reproducible manner. BOSH supports deployments across different IaaS providers, such as VMware vSphere, Google Compute Platform, and Amazon Elastic Compute Cloud (EC2).

The command-line interface and API of BOSH support multiple use cases through the lifecycle of Kubernetes. You can deploy multiple Kubernetes cluster in minutes. Scaling Kubernetes clusters can also be done with CLI or API calls. Patching and updating one or more Kubernetes clusters are also made easier by PKS through the same mechanism, making sure your clusters always keep pace with the latest security and maintenance updates. If the clusters are no longer required, the user can quickly delete them.

Container Networking with VMware NSX

VMware NSX-T supplies Kubernetes clusters with advanced container networking and security features, such as micro-segmentation, load balancing, ingress control, and security policies. NSX furnishes the complete set of Layer 2 through Layer 7 networking services that is needed for pod-level networking in Kubernetes. You can quickly deploy networks with micro-segmentation and on-demand network virtualization for containers and pods.

The integration of NSX with PKS delivers an immediate, far-reaching impact on network operations for cloud-native applications:

- The native support for NSX-T load balancers provides highly reliable, high-performance distribution of traffic to Kubernetes services that are exposed externally.

- Policies for micro-segmentation that go beyond the standard security policies of Kubernetes.
- Network polices that help secure traffic across Kubernetes namespaces and within pods in the same namespace.
- Operational tools and troubleshooting utilities that can debug inter-pod communication.
- A unified policy layer for VMs and Kubernetes pods.

In PKS, NSX-T automates container networking in Kubernetes. An app running in the Kubernetes cluster can use the virtual network to communicate with the outside world. Incoming traffic makes use of the load balancer, which NSX automatically provisioned for the Kubernetes cluster.

When a cluster is created on PKS, NSX dynamically creates a secured network for the Kubernetes cluster nodes. NSX-T load balancing services are on a highly available, redundant NSX Edge cluster, so if one load balancer goes down, traffic automatically falls over to another load balancer. The load balancing services are integrated with the Kubernetes Ingress and LoadBalancer constructs.

Network Policies and Micro-Segmentation

NSX adds network policies and micro-segmentation to meet the isolation requirements of workloads. You can, for example, define micro-segmentation policies based on traffic flow patterns among the namespaces in which containerized applications are running. Network policies can also segregate pods to securely handle a microservices-based architecture. Each Kubernetes namespace can be isolated from other namespaces. If you have three namespaces, for example, NSX automatically sets up an isolated network for each one. With NSX managing container networking interfaces on PKS, network policies specify how traffic can move both between and within Kubernetes namespaces. In short, NSX lets you craft rules to impose your security requirements on workloads.

NSX can enforce additional types of policies:

- Group policies based on IP address
- Egress policies
- Policies that route traffic to to different virtual machines based on the names of VMs.
- Policies that specify what traffic can enter and leave the network for a containerized application.

Another powerful result of NSX integration with PKS is an assortment of operational tools and troubleshooting utilities:

- Traceflow
- Port mirroring
- Port connection tool
- Spoofguard
- Syslog
- Port counters
- IPFIX

Such tools are the mainstay of a modern, virtualized network. And now they have been ported to container networking on Kubernetes. Such tools fulfill the requirements of production-level networking for containerized applications so you can, for example, debug communication between pods and the microservices components of your containerized applications.

A Boon to Operations

In these ways, NSX supplies an industrial-strength, production-grade solution for container networking. So what's the result of all this for operations?

First off, because NSX automates provisioning for container interfaces, it frees developers from having to request networking infrastructure from IT, and it frees IT from having to fulfill those requests. The result: No more provisioning bottlenecks. The inefficiencies attached to the manual process of fulfilling infrastructure requests evaporate in a wave of automation. Another result: A boost in developers' productivity: They no longer need to fuss about with submitting tickets to obtain the resources they constantly need.

But there's more. Because NSX provides secure networking for microservices-based applications running on Kubernetes, developers can rapidly, frequently, and confidently deploy software without having to write code to guard against traditional infrastructure issues.

There are other results as well, all critical outcomes associated with moving in the direction of cloud-native applications:

- Being able to modernize legacy applications more quickly and efficiently.

- Being able to modify existing applications faster, with less effort, and with more predictability.
- Being able to deploy and redeploy applications with enhanced flexibility, agility, predictability, and repeatability.

Secure Image Registry from Project Harbor

Harbor is an open source, enterprise-class registry server from VMware that stores and distributes Docker images in a private registry behind your firewall. Harbor includes role-based access control, vulnerability scanning for container images, policy-based image replication, and notary and auditing services.

Integration with LDAP or Microsoft Active Directory ensures the proper level of authority and access for container images.

The image notary service establishes content trust by letting publishers sign images when they push them and preventing unsigned images from being pulled.

With the private registry, users can scan container images for vulnerabilities to mitigate the risk of security breaches related to contaminated container images.

Persistent Storage

PKS allows customers to deploy Kubernetes clusters for both stateless and stateful applications. It supports the vSphere Cloud Provider storage plugin which is part of Kubernetes through Project Hatchway. This allows PKS to support Kubernetes storage primitives such as Volumes, Persistent Volumes (PV), Persistent Volumes Claims (PVC), Storage Class and Stateful Sets on vSphere storage, and also brings in enterprise-grade storage features like Storage Policy Based Management(SPBM) with vSAN to Kubernetes based applications.

VMware vSphere Cloud Provider: To run stateful, data-intensive containerized applications that include databases, you need a persistent storage solution. vSphere Cloud Provider, which is part of Kubernetes through Project Hatchway, enables PKS to support Kubernetes the following storage primitives on vSphere storage: Volumes, Persistent Volumes (PV), Persistent Volumes Claims (PVC), Storage Class, and Stateful Sets. The vSphere Cloud Provider also furnishes enterprise storage features like storage policy-based management.

Managing Operations by Integrating with Other VMware Solutions

PKS can be integrated with other VMware products to offer a full-stack Kubernetes service. Here are some of the VMware products with which PKS can integrate:

VMware vRealize® Operations™: With vRealize Operations, IT organizations can improve performance, avoid business disruption, and become more efficient with comprehensive visibility across applications and infrastructure.

VMware vRealize Log Insight™: Log Insight delivers highly scalable log management with actionable dashboards, analytics, and broad third-party extensibility, giving you deep operational visibility and faster troubleshooting.

Wavefront® by VMware: Wavefront efficiently monitors containers at scale. Its dashboards give DevOps real-time visibility into the operations and performance of containerized workloads and Kubernetes clusters.

High Availability

PKS provides critical production-grade capabilities to ensure maximum uptime for workloads running in your Kubernetes clusters. It continuously monitors the health of all underlying VM instances, and recreates VMs when there are failed or unresponsive nodes. It also manages the rolling upgrade process for a fleet of Kubernetes clusters, allowing clusters to be upgraded with no downtime for application workloads.

Constant Compatibility with Google Kubernetes Engine (GKE)

PKS is developed using mainline Kubernetes and delivers the latest stable Kubernetes release to your developers. It ensures constant compatibility with Kubernetes versions that are supported by GKE, so enterprise developers can use the latest features and patches across vSphere and GKE. In addition, without adding any proprietary abstraction layer on top of Kubernetes, PKS exposes Kubernetes in its native form, letting developers or your development tools interact with Kubernetes using the native Kubernetes interface, and also making workloads readily portable between vSphere and GKE.

Multi-Tenancy

To isolate workloads and ensure privacy, PKS supports multi-tenancy for multiple lines of business within an enterprise. Different users or different lines of business are able to use their own Kubernetes clusters. Additionally, with NSX-T micro-segmentation, Kubernetes namespaces can be secured for multiple teams using a shared cluster.

Multi-Cloud

PKS supports multi-cloud deployment through BOSH. With PKS, you can deploy containerized application with Kubernetes on-premises on vSphere, or on public clouds such as Google Cloud Platform.

Summary of PKS Features and Benefits

On-Demand Provisioning	Accelerates the deployment of Kubernetes clusters. Eliminates manual steps for deploying Kubernetes clusters. Minimizes mistakes and shortens time-to-value
On-Demand Scaling	Scales the cluster capacity easily. Eliminates manual steps and mistakes. Optimizes resource utilization
On-Demand Patching	Centralizes and speeds up patching and updating of multiple Kubernetes clusters. Keeps Kubernetes cluster up-to-date and secure.
Rolling Upgrades	Minimizes workload downtime by rolling upgrading a fleet of Kubernetes clusters.
Automatic Health Check and Self-Healing	Prevents issues with proactive monitoring of the health of all nodes. Ensures desired responsiveness of the application services by recreating failed or unresponsive nodes.
Advanced Container Networking and Security	Increases developer and ops productivity by simplifying networking management and enhancing security. Optimizes native container networking including automatic provisioning, micro-segmentation, ingress controller, load balancing and security policies.

Secure Container Registry:	Minimizes application breaches with enhanced container security. Simplifies container image management and enhances security through image replication, RBAC, AD/LDAP integration, notary services, vulnerability scanning, and auditing.
Constant Compatibility with GKS	Enhances developer productivity by letting developers access the most up-to-date Kubernetes features and tools.
Native Kubernetes Support	Exposes Kubernetes in its native form with no proprietary extensions. Increases developer productivity by offering them the native Kubernetes CLI and full YMAL support.
CNCF-certified Kubernetes Distribution	Compliant with the community's specification. Ensures portability, interoperability and consistency between different environments cross-cloud.
Multi-tenancy	Provides individual users with their own Kubernetes cluster on isolated network. Secures workloads between tenants and provides privacy.
Persistent Storage	Deploys Kubernetes clusters for both stateless and stateful applications. Supports vSphere Cloud Provider storage plugin through Project Hatchway.
Multi-cloud	Optimizes workload deployment in multi-cloud environments by providing a consistent interface to deploy and manage Kubernetes on both vSphere and Google Cloud Platform.
Integration with vRealize Operations	Increases operations efficiency by letting IT administrators effectively monitor and troubleshoot the performance of the Kubernetes clusters and its underlying infrastructure.
Integration with Wavefront by VMware	Offers near real-time visibility into the operations and performance of containerized applications running in the Kubernetes clusters. Allows developers and DevOps to do Application Performance Monitoring and Management (APM).
Integration with vRealize Log Insight	Delivers highly scalable log management with actionable dashboards, analytics, and broad third-party extensibility. Enables deep operational visibility and faster troubleshooting.

Use Cases

This chapter looks at several common use cases for vSphere Integrated Containers and VMware Pivotal Container Service.

Self-Service Infrastructure for Agile Development

Deploying infrastructure to support application development is often cumbersome, error-prone, and time-consuming. As developers rush to build new apps, IT teams waste time with manual configuration, provisioning, and scripting. To improve productivity, they need, at the very least, to streamline the way they roll out and manage infrastructure.

A key solution is providing developers with a sandbox so they can serve their own infrastructure needs by creating Docker container hosts on demand. Modern developers need an environment where they can build and run their apps using native container technology with minimal involvement from IT. Implementing a developer sandbox by using VMware vSphere® Integrated Containers™ provides developers with an agile self-service container environment for app development.

Developers are increasingly turning to Docker containers because containers help them adapt to changes brought about by digital transformation. The architecture of a containerized application complements agile practices and DevOps methodologies, such as continuous integration and continuous delivery.

Supporting Microservices

Developers often turn to container technology to support micro-services. A micro-services architecture breaks up the functions of an application into a set of small, discrete, decentralized, goal-oriented processes, each of which can be independently developed, tested, deployed, replaced, and scaled.

However, trying to build an application with micro-services on a laptop or desktop can hit performance and memory constraints. Even when an application does not use micro-services, a laptop might not have enough resources. Whenever developers don't have enough resources on their laptops to run a copy of their production environment, a sandbox enables developers to work on their app.

Providing a Developer Sandbox with vSphere Integrated Containers

vSphere Integrated Containers creates an enterprise container infrastructure within vSphere, enabling both traditional and containerized apps to run side by side on a common infrastructure. Developers can initiate Docker container hosts within a resource pool so they can spin containers up and down on demand without having to file a ticket with IT.

Self-Service Provisioning

Developers can self-provision Docker container hosts. Although this ticketless environment gives developers the Docker tools they need to build modern applications or repackage existing ones in containers, IT retains governance and control over the infrastructure because vSphere Integrated Containers leaves the management of the hosts to the vSphere administrator.

vSphere administrators provision compute, networking, and storage resources and provide them to tenants as a self-service portal exposing a familiar Docker-compatible API. The virtual machines provisioned using vSphere Integrated Containers take advantage of many of the benefits of vSphere, including Distributed Resource Scheduler, clustering, VMware vSphere vMotion®, VMware vSphere High Availability (HA), distributed port groups, and shared storage.

Developers and DevOps need not worry about patching, security, isolation, tenancy, availability, clustering, or capacity planning. Those functions continue to be business as usual for the vSphere administrator. Instead, developers and DevOps receive a container endpoint as a service. The outcome is a win-win situation for both developers and administrators: The vSphere administrator gets visibility into and control over the virtual machines, while developers and DevOps can self-provision Docker container hosts and work with them by using a Docker client.

Because of the portability of the Docker image format, a developer using vSphere Integrated Containers can establish an endpoint at the end of a continuous integration pipeline, consuming images pushed to the private, secure registry that comes with vSphere Integrated Containers. There is no need to build out a separate, dedicated container infrastructure stack. The finished application can be put into production on a virtual container host powered by vSphere Integrated Containers.

Sharing Images with the Private Registry

A developer can also push a container image for an application being developed to the vSphere Integrated Containers registry, tag it, and let other developers use a Docker client to run the container on a Docker container host. At the same time, a vCenter administrator can see each Docker container host in the vSphere inventory. The developer or the administrator can use the monitoring page in the vSphere Integrated Containers management portal to view statistics and logs about containers. The management portal is integrated with identity management to securely provision containers.

Docker Container Hosts on Demand

Developers can exploit the capacity of a VMware® software-defined data center to develop and test a containerized application. A laptop might be too sluggish to run a containerized application, especially if it is built with microservices. With vSphere Integrated Containers, developers can quickly provision Docker container hosts and then point their Docker client to the host to work with containers. A developer sandbox powered by vSphere Integrated Containers lets developers and DevOps serve their own requirements by creating Docker container hosts on demand. The outcome accelerates the process of developing software and shortens an application's time to market.

Repackaging an Application with VIC

Digital transformation is fundamentally disrupting how software is developed and deployed. Companies are under pressure to rapidly create innovative software that engages their customers and provides new services. Improving time to market is paramount. As a result, companies are turning to container technology to modernize their data centers and streamline software development.

Containers package an application and its dependencies into a distributable image that can run almost anywhere. The packaging and portability of containers support modern architectural patterns and make developers more efficient.

By provisioning and hosting containers, VMware vSphere Integrated Containers prepares your data center for the digital era. The solution moves you one step closer to a modernized software-defined data center that

deploys infrastructure, services, data, and applications on demand. For a traditional application, however, a common first step toward modernization is repackaging part or all of it in a container.

Challenges Impeding Application Repackaging

Most infrastructure platforms are not designed to run traditional and modern applications side by side while working with existing hardware and software, making it difficult to repackage a traditional application with containers.

Beyond the infrastructure, modern applications pose their own challenges. Modern apps change frequently, are developed in short release cycles, and might be built with microservices. In addition, IT teams need to connect applications across clouds and devices with security, compliance, and availability.

But you can establish a consistent operational model for infrastructure and application delivery that works with both traditional and containerized applications. The model creates a powerful bridge to move from traditional software development practices to new, more flexible forms geared toward innovation, speed of execution, and easier maintenance.

Repackaging Applications for Efficiency

THE BENEFITS OF REPACKAGING APPS IN A CONTAINER
Advantages of repackaging a traditional app in a container and running it with vSphere Integrated Containers:

- Ease application maintenance
- Minimize disruption to operations and reduce costs by using existing VMware infrastructure
- Simplify workflows to accelerate development
- Fix an application's vulnerabilities
- Impose a consistent environment across development, testing and production
- Enhance portability
- Streamline app deployment by using Docker
- Improve the app's time to market

It can be costly and time-consuming to re-architect an in-house application that is too coupled to its data or other application components. For an application with a well-defined architecture that tightly couples data with application logic, it makes sense to repackage the application in a container without having to modify the application's design. In addition, the learning curve for repackaging an application or part of it, such as the web front end, is small.

vSphere Integrated Containers provides an alternate way to instantiate a Docker image by letting you use the Docker command-line interface and then deploy the container image as a VM instead of as a container on top of a Docker host. As a result, you reap the benefits of packaging the application as a container without re-architecting it. This approach keeps the isolation benefits of VMs.

vSphere Integrated Containers is ideally suited to application repackaging. No new infrastructure or dedicated hardware is required to repackage the application, nor do you need to implement new tooling. The repackaged containerized application can run alongside other virtual machines running other applications, whether traditional or containerized. To support the repackaged container, vSphere Integrated Containers provides high availability at the infrastructure level without developer intervention. You can also use such core vSphere features as vSphere High Availability and vSphere vMotion.

Containers simplify application maintenance. After you repackage an app in a container, maintenance activities such as upgrading, updating, and patching become easier. The Docker file, in particular, eases patches and upgrades.

Unifying Containerized Applications with vSphere

Docker furnishes a platform with which developers can rapidly build applications on their laptops and then port them to vSphere Integrated Containers. A developer working on a traditional Java application running on Apache Tomcat, for example, can containerize the application and then, because of its inherent portability, shift it to a virtual container host provisioned by a vSphere administrator.

The developer can then push the container image to the vSphere Integrated Containers registry, tag it, and run it in the virtual container host. At the same time, an administrator of VMware vCenter® can see the con-

tainer VM in the vSphere inventory. The developer or the administrator can use the monitoring page in the vSphere Integrated Containers portal to view statistics and logs. This unification is made possible in part by the vSphere Integrated Containers management portal, which is integrated with identity management to securely provision containers. The result enables application development teams to repackage, test, and deploy applications quickly and efficiently.

Replatforming Applications with PKS

Repackaging an application to run in containers and then moving the app to a modern platform—that is, replatforming the app—is a critical step toward reaping the benefits of container technology. Replatforming accelerates software development, eases infrastructure management, and automates deployment. After deployment, a replatformed application can be orchestrated and scaled on demand with Kubernetes. The power of Kubernetes to orchestrate containerized workloads is key to unlocking the benefits of replatforming an application.

Benefits of Replatforming

Replatforming an application propels you toward several objectives associated with accelerating application development and deployment without having to deal with the complexity of re-architecting or refactoring an application:

- Workload consolidation, especially if you are increasingly moving in the direction of developing cloud-native applications.
- Simplified and improved integration with a continuous integration and continuous deployment pipeline (CI/CD).
- Operational efficiency for managing the application with automation, security,monitoring, logging, analytics, and lifecycle management.

Because replatforming takes place after repackaging an application in containers,you also reap the benefits of repackaging:

- Portability across development, test, production, and cloud environments.
- Predictability and reproducibility to eliminate the it-worked-for-me refrain.
- Simplicity of upgrading, patching, and maintenance.
- Velocity for agile development iterations, testing, and deployment.

- Flexibility for developers to code where and when they want with the tools they like.
- Traceability of immutable container images for improved transparency,compliance, and reuse.

Replatforming an app also puts you in a position to take advantage of changes in ISV-delivered applications—which are increasingly being prepackaged with their dependencies in containers for a consistent, problem-free installation process.

Targeting Workloads for Replatforming on PKS

With its flexible, powerful capabilities, VMware PKS is well suited to replatforming the following types of workloads:

- Applications requiring data persistence, such as MongoDB, CouchDB,and Elasticsearch.
- Applications managed as a distributed cluster, especially when nodes in the cluster must communicate with one another.
- Applications that need infrastructure primitives, such as persistent storage.
- Applications that require multiple ports.PKS delivers services that empower developers to manage their container images with the built-in registry, to build container and pod templates for Kubernetes, to configure the port bindings that they want, and to manage dependencies. As such,PKS is ideal for replatforming modern data services such as Elasticsearch, Spark, and other applications requiring a custom stack or access to infrastructure primitives.

Decomposing the Monolith in Stages

After replatforming an application on VMware PKS, you can separate it into three components in stages. During the first stage, the database can be decoupled from the monolith so that it can be independently scaled. During the second stage, the application's front end, including its user interface and command-line interface,can be detached so it can be managed and updated separately. The third stage focuses on security to make sure that inter-component communication is secure.

Section Summary

VMware Pivotal Container Services delivers a highly available, production-grade Kubernetes-based container service equipped with container networking, security,and lifecycle management. Deployable both on-prem in vSphere and in public clouds like Google Cloud Platform, VMware PKS is well suited to replatforming applications that will benefit from containerization and orchestration.

Deploying New Cloud-Native Apps with PKS

If you are seeking to build new cloud-native applications, PKS furnishes a flexible, scalable Kubernetes-based container service that simplifies deployment and operations. With PKS, developers can provide container images and templates for pods. At the same time, the platform provides the flexibility for customization—developers can, for example, set up explicit port bindings for containers, co-locate them, and configure routes and dependencies. For flexibility in managing and automating containers, PKS exposes the Kubernetes API.

DevOps or a platform operations team is likely to play a key role in managing PKS and in providing a system and tools for continuous delivery, such as Jenkins, a pipeline automation tool.

Here are the some of the ideal use cases and workloads for PKS:

- Running modern data services such as Elasticsearch, Cassandra, and Spark.
- Running ISV applications packaged in containers.
- Running microservices-based apps that require a custom stack.

Exploiting the Power of Containers

This chapter presents detailed scenarios and examples of how to deploy and adopt various container technology to exploit the power of containers.

Running a Containerized App with Photon OS on Amazon Elastic Cloud Compute

This section introduces you to working with a containers in the cloud by demonstrating how to use a Linux container host to launch a containerized application. The section describes how to get Photon OS up and running on Amazon Web Services Elastic Cloud Compute (EC2), customize Photon with cloud-init, connect to it with SSH, and run a containerized application.

Photon OS is an open source Linux container host optimized for cloud-native applications, cloud platforms, and VMware infrastructure. Photon OS provides a secure run-time environment for efficiently running containers. For an overview of Photon OS, see https://vmware.github.io/photon/.

Prerequisites

Using Photon OS within AWS EC2 requires the following resources:

- AWS account. Working with EC2 requires an Amazon account for AWS with valid payment information. Keep in mind that, if you try the examples in this document, you will be charged by Amazon. See Setting Up with Amazon EC2.
- Amazon tools. The following examples also assume that you have installed and configured the Amazon AWS CLI and the EC2 CLI and AMI tools, including ec2-ami-tools.

See Installing the AWS Command Line Interface, Setting Up the Amazon EC2 Command Line Interface Tools on Linux, and Configuring AWS Command-Line Interface. Also see Setting Up the AMI Tools. This article uses an Ubuntu 14.04 workstation to generate the keys and certificates that AWS requires.

Downloading the Photon OS Image for Amazon

VMware packages Photon OS as a cloud-ready Amazon machine image (AMI) that you can download for free from Bintray.

Download the Photon OS AMI now and save it on your workstation. For instructions, see Downloading Photon OS.

Note: The AMI version of Photon is a virtual appliance with the information and packages that Amazon needs to launch an instance of Photon in the cloud. To build the AMI version, VMware starts with the minimal version of Photon OS and adds the sudo and tar packages to it.

Getting Photon OS Up and Running on EC2

To run Photon OS on EC2, you must use cloud-init with an EC2 data source. The cloud-init service configures the cloud instance of a Linux image. An *instance* is a virtual server in the Amazon cloud.

The examples in this article show how to generate SSH and RSA keys for your Photon instance, upload the Photon OS .ami image to the Amazon cloud, and configure it with cloud-init. In many of the examples, you must replace information with your own paths, account details, or other information from Amazon.

Step 1: Create a Key Pair

The first step is to generate SSH keys on, for instance, an Ubuntu workstation:

```
ssh-keygen -f ~/.ssh/mykeypair
```

The command generates a public key in the file with a .pub extension and a private key in a file with no extension. Keep the private key file and remember the name of your key pair; the name is the file name of the two files without an extension. You'll need the name later to connect to the Photon instance.

Change the mode bits of the public key pair file to protect its security. In the command, include the path to the file if you need to.

```
chown 600 mykeypair.pub
```

Change the mode bits on your private key pair file so that only you can view it:

```
chmod 400 mykeypair
```

To import your public key pair file (but not your private key pair file), connect to the EC2 console at https://console.aws.amazon.com/ec2/ and select the region for the key pair. A key pair works in only one region, and the instance of Photon that will be uploaded later must be in the same region as the key pair. Select key pairs under Network & Security, and then import the public key pair file that you generated earlier.

For more information, see Importing Your Own Key Pair to Amazon EC2.

Step 2: Generate a Certificate

When you bundle up an image for EC2, Amazon requires an RSA user signing certificate. You create the certificate by using openssl to first generate a private RSA key and then to generate the RSA certificate that references the private RSA key. Amazon uses the pairing of the private key and the user signing certificate for handshake verification.

First, on Ubuntu 14.04 or another workstation that includes openssl, run the following command to generate a private key. If you change the name of the key, keep in mind that you will need to include the name of the key in the next command, which generates the certificate.

```
openssl genrsa 2048 > myprivatersakey.pem
```

Remember where you store your private key locally; you'll need it again later.

Second, run the following command to generate the certificate. The command prompts you to provide more information, but because you are generating a user signing certificate, not a server certificate, you can just type Enter for each prompt to leave all the fields blank.

```
openssl req -new -x509 -nodes -sha256 -days 365 -key mypri-
vatersakey.pem -outform PEM -out certificate.pem
```

For more information, see the Create a Private Key and the Create the User Signing Certificate sections of Setting Up the AMI Tools.

Third, upload to AWS the certificate value from the certificate.pem file that you created in the previous command. Go to the Identity and Access Management console at https://console.aws.amazon.com/iam/, navigate to the name of your user, open the Security Credentials section, click Manage Signing Certificates, and then click Upload Signing Certificate.

Open certificate.pem in a text editor, copy and paste the contents of the file into the Certificate Body field, and then click Upload Signing Certificate.

For more information, see the Upload the User Signing Certificate section of Setting Up the AMI Tools.

Step 3: Create a Security Group

The next prerequisite is to create a security group and set it to allow SSH, HTTP, and HTTPS connections over ports 22, 80, and 443, respectively. Connect to the EC2 command-line interface and run the following commands:

```
aws ec2 create-security-group --group-name photon-sg
--description "My Photon security group"
{
    "GroupId": "sg-d027efb4"
}
aws ec2 authorize-security-group-ingress --group-name pho-
ton-sg --protocol tcp --port 22 --cidr 0.0.0.0/0
```

The GroupId is returned by EC2. Write it down; you'll need it later.

By using 0.0.0.0/0 for SSH ingress on Port 22, you are opening the port to all IP addresses–which is not a security best practice but a convenience for the examples in this article. For a production instance or other instances that are anything more than temporary machines, you should authorize only a specific IP address or range of addresses. See Amazon's document on Authorizing Inbound Traffic for Linux Instances.

Repeat the command to allow incoming traffic on Port 80 and on Port 443:

```
aws ec2 authorize-security-group-ingress --group-name pho-
ton-sg --protocol tcp --port 80 --cidr 0.0.0.0/0

aws ec2 authorize-security-group-ingress --group-name pho-
ton-sg --protocol tcp --port 443 --cidr 0.0.0.0/0
```

Check your work:

```
aws ec2 describe-security-groups --group-names photon-sg
```

Step 4: Extract the Tarball

Next, make a directory to store the image, and then extract the Photon OS image from its archive by running the following tar command. (You might have to change the file name to match the version you have.)

```
mkdir bundled
tar -zxvf ./photon-ami.tar.gz
```

Step 5: Bundle the Image

The next step is to run the ec2-bundle-image command to create an instance store-backed Linux AMI from the Photon OS image that you extracted in the previous step. The result of the ec2-bundle-image command is a manifest that describes the machine in an XML file.

The command uses the certificate path to your PEM-encoded RSA public key certificate file; the path to your PEM-encoded RSA private key file; your EC2 user account ID; the correct architecture for Photon OS; the path to the Photon OS AMI image extracted from its tar file; and the bundled directory from the previous step.

You must replace the values of the certificate path, the private key, and the user account with your own values.

```
$ ec2-bundle-image --cert certificate.pem --privatekey
myprivatersakey.pem --user <EC2 account id>  --arch x86_64
--image photon-ami.raw --destination ./bundled/
```

Step 6: Put the Bundle in a Bucket

Next, make an S3 bucket, replacing <bucket-name> with the name that you want. The command creates the bucket in the region specified in your Amazon configuration file, which should be the same region in which you are using your key pair file:

```
$ aws s3 mb s3://<bucket-name>
```

Now upload the bundle to the Amazon S3 cloud. The following command includes the path to the XML file containing the manifest for the Photon OS machine created during the previous step, though you might have to change the file name to match the version you have. The manifest file is typically located in the same directory as the bundle.

The command also includes the name of the Amazon S3 bucket in which the bundle is to be stored; your AWS access key ID; and your AWS secret access key.

```
$ ec2-upload-bundle --manifest ./bundled/photon-ami.mani-
fest.xml --bucket <bucket-name> --access-key <Account Access
Key> --secret-key <Account Secret key>
```

Step 7: Register the Image

The final step in creating an AMI before you can launch it is to register it. The following command includes a name for the AMI, its architecture, and its virtualization type. The virtualization type for Photon OS is hvm.

```
$ ec2-register <bucket-name>/photon-ami.manifest.xml --name
photon-ami --architecture x86_64 --virtualization-type hvm
```

Once registered, you can launch as many new instances as you want.

Step 8: Run an Instance of the Image with Cloud-Init

Now things get a little tricky. In the following command, the user-data-file option instructs cloud-init to import the cloud-config data in user-data.txt.

The command also includes the ID of the AMI, which you can obtain by running ec2-describe-images; the instance type of m3.medium, which is a general purpose instance type; and the name of key pair, which should be replaced with your own–otherwise, you won't be able to connect to the instance.

Before you run the command, change directories to the directory containing the mykeypair file and add the path to the user-data.txt.

```
$ ec2-run-instances <ami-ID> --instance-type m3.medium -g
photon-sg --key mykeypair --user-data-file user-data.txt
```

Here are the contents of the user-data.txt file that cloud-init applies to the machine the first time it boots up in the cloud.

```
#cloud-config
hostname: photon-on-01
groups:
- cloud-admins
- cloud-users
users:
- default
- name: photonadmin
   gecos: photon test admin user
   primary-group: cloud-admins
   groups: cloud-users
   lock-passwd: false
   passwd: vmware
- name: photonuser
   gecos: photon test user
   primary-group: cloud-users
   groups: users
```

```
   passwd: vmware
packages:
- vim
```

Step 9: Get the IP Address of Your Image

Now run the following command to check on the state of the instance that you launched:

```
$ ec2-describe-instances
```

Finally, you can obtain the external IP address of the instance by running the following query:

```
$ aws ec2 describe-instances --instance-ids <instance-id>
--query 'Reservations[*].Instances[*].PublicIpAddress'
--output=text
```

If need be, check the cloud-init output log file on EC2 at /var/log/cloud-init-output.log to see how EC2 handled the settings in the cloud-init data file.

For more information on using cloud-init user data on EC2, see Running Commands on Your Linux Instance at Launch.

Deploy a Containerized Application in Photon OS

This section shows you how to connect to the Photon instance by using SSH and to launch a web server by running it in Docker.

Step 1: Connect with SSH

Connect to the instance over SSH by specifying the private key (.pem) file and the user name for the Photon machine, which is root:

```
ssh -i ~/.ssh/mykeypair root@<public-ip-address-of-instance>
```

For complete instructions, see Connecting to Your Linux Instance Using SSH.

Step 2: Run Docker

On the minimal version of Photon OS, the docker engine is enabled and running by default, which you can see by running the following command:

```
systemctl status docker
```

Step 3: Start the Web Server

Note: Please make sure that the proper security policies have been enabled on the Amazon AWS side to enable traffic to port 80 on the VM.

Since Docker is running, you can run an application in a container–for example, the Nginx Web Server. This example uses the popular open source web server Nginx. The Nginx application has a customized VMware package that the Docker engine can download directly from the Docker Hub.

To pull Nginx from its Docker Hub and start it, run the following command:

```
docker run -p 80:80 vmwarecna/nginx
```

The Nginx web server should be bound to the public DNS value for the instance of Photon OS–that is, the same address with which you connected over SSH.

Step 4: Test the Web Server

On your local workstation, open a web browser and go to the the public address of the Photon OS instance running Docker. The following screen should appear, showing that the web server is active:

Figure 14: Nginx

When you're done, halt the Docker container by typing Ctrl+c in the SSH console where you are connected to EC2.

You can now run other containerized applications from the Docker Hub or your own containerized application on Photon OS in the Amazon cloud.

Launching the Web Server with Cloud-Init

To eliminate the manual effort of running Docker, you can add docker run and its arguments to the cloud-init user data file by using runcmd:

```
#cloud-config
hostname: photon-on-01
groups:
- cloud-admins
- cloud-users
users:
- default
- name: photonadmin
   gecos: photon test admin user
   primary-group: cloud-admins
   groups: cloud-users
   lock-passwd: false
   passwd: vmware
- name: photonuser
   gecos: photon test user
   primary-group: cloud-users
   groups: users
   passwd: vmware
packages:
- vim
runcmd:
- docker run -p 80:80 vmwarecna/nginx
```

To try this addition, you'll have to run another instance with this new cloud-init data source and then get the instance's public IP address to check that the Nginx web server is running.

Terminating the AMI Instance

Because Amazon charges you while the instance is running, make sure to shut it down when you're done.

First, get the ID of the AMI so you can terminate it:

```
$ ec2-describe-instances
```

Finally, terminate the Photon OS instance by running the following command, replacing the placeholder with the ID that the ec2-describe-images command returned. If you ran a second instance of Photon OS with the cloud-init file that runs docker, terminate that instance, too.

```
$ ec2-terminate-instances <instance-id>
```

Integrating Lightwave with Photon OS

Lightwave provides security services to Photon OS. You can use Lightwave to join a Photon OS virtual machine to the Lightwave directory service and then authenticate users with Kerberos.

Because the Photon OS repository includes the Lightwave packages, installing the packages for either Lightwave client or the server is simple. Here's an example of installing the Lightwave server packages on a virtual machine running Photon OS:

```
tdnf install vmware-lightwave-server
Installing:
apache-ant  noarch  1.10.1-1.ph2dev  3.66 M
vmware-dns-client  x86_64  1.2.0-1.ph2dev  614.42 k
apache-tomcat  noarch  8.5.13-2.ph2dev  8.59 M
commons-daemon  x86_64  1.0.15-9.ph2dev  79.41 k
jansson  x86_64  2.10-1.ph2dev  74.52 k
vmware-sts-client  x86_64  1.2.0-1.ph2dev  41.09 M
vmware-sts  x86_64  1.2.0-1.ph2dev  67.91 M
vmware-afd  x86_64  1.2.0-1.ph2dev  763.81 k
vmware-dns  x86_64  1.2.0-1.ph2dev  344.19 k
vmware-directory  x86_64  1.2.0-1.ph2dev  4.03 M
vmware-ca-client  x86_64  1.2.0-1.ph2dev  501.53 k
vmware-ic-config  x86_64  1.2.0-1.ph2dev  114.89 k
likewise-open  x86_64  6.2.11-1.ph2dev  11.26 M
vmware-afd-client  x86_64  1.2.0-1.ph2dev  931.89 k
vmware-directory-client  x86_64  1.2.0-1.ph2dev  714.02 k
vmware-ca  x86_64  1.2.0-1.ph2dev  206.27 k
vmware-lightwave-server  x86_64  1.2.0-1.ph2dev  0.00 b
Total installed size:  140.78 M
```

(In the names of the packages, "afd" stands for authentication framework daemon; "ic" stands for infrastructure controller, which is Lightwave's internal name for its domain controller. Several of the packages, such as Jansson and Tomcat, are used by Lightwave for Java services or other tooling.)

The convenience and expedience of being able to instantly install the Lightwave packages from a secure, signed VMware repository become even more significant when the cloud-ready image of Photon OS runs on Amazon Elastic Cloud Compute or Google Compute Engine. The Amazon machine image of Photon OS and the Google Compute Engine version of Photon OS are available as free downloads on Bintray.

Deploying Lightwave on AWS

Lightwave can run on Photon OS on Amazon Elastic Compute Cloud to provide identity services to your machines, users, and applications running in the Amazon cloud. The process of deploying Lightwave on EC2 entails creating a Photon OS instance, setting firewall rules to open several ports, setting a hostname for the machine, installing the Lightwave server components, and promoting a Lightwave domain controller. Once deployed, the Lightwave domain controllers appear in the EC2 Dashboard:

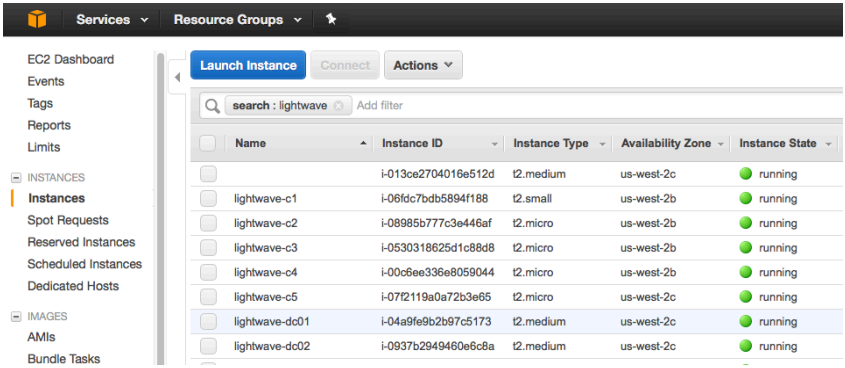

Figure 15: Lightwave domain controllers running on Amazon EC2.

For more information, see Lightwave on GitHub.

Deploying Lightwave on Google

Cloud administrators and DevOps personnel can rapidly deploy Lightwave on Google Compute Engine by using Photon OS or another Linux image, such as Ubuntu. The process goes like this:

- Set up firewall rules and open ports for Lightwave DNS, LDAP, STS, and the other Lightwave services.
- Upload the freely available Photon OS image for GCE.
- Create a Photon OS instance, set the hostname for your Lightwave instance, and set the instance to use Lightwave for DNS.
- Install Lightwave from the Photon OS repository.
- Promote the first Lightwave domain controllers and add more of them if you want.

For instructions on how to set up Lightwave on GCE, see Lightwave on GitHub.

After deploying and promoting the Lightwave domain controllers, you can see them in the GCE web interface:

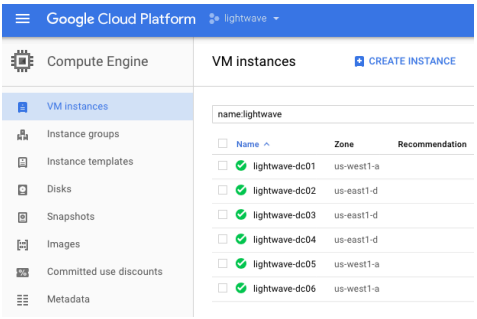

Figure 16: Lightwave domain controllers running on Google Cloud Platform.

Using vSphere Integrated Containers to Solve Container Networking Problems

The image below shows a high-level view of the networks that vSphere Integrated Containers (VIC) use and how they connect to your VMware vSphere environment, the Registry and Management Portal and to public registries, such as Docker Hub.

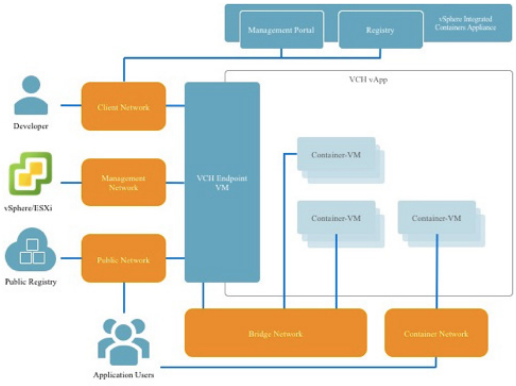

Figure 17: vSphere Integrated Containers networking.

As you can see from the picture above, a Virtual Container Host (VCH) not only allows you to easily segregate management traffic from data traffic, but also Docker client traffic from intra-container traffic. Moreover, since containers in VIC are deployed as virtual machines (VMs), vSphere administrators can make vSphere networks directly available to containers.

Networking Options

The VIC network overview lightboard details the networking concepts for vSphere Integrated Containers, while the recently updated documentation comes in handy to further explain these options:

- Client Network: The Client Network is used by a VCH to expose the Docker API service and where developers must point their Docker clients to manage and run containers.
- Public Network: The Public Network is used by a VCH to pull images from registries. The most common use case is to pull images from the public Docker hub. You can also create your own private, secure local registry by using the VIC Registry (based on Project Harbor).
- Management Network: The Management Network is used by a VCH to securely communicate with vCenter and ESXi hosts.
- Bridge Network: The Bridge Network is a private network for container communication. External access is granted by exposing ports to containers and routing the traffic through the VCH endpoint VM. With no extra configuration, VIC provides service discovery while a built-in IPAM server provides the containerVMs with private IP addresses from the subnet of the bridge network.
- Container Network: A Container Network is a user-defined network that can be used to connect containerVMs directly to a routable network. Container networks allow vSphere administrators to make vSphere networks directly available to containers. Container networks are specific to VIC and have no equivalent in Docker.

For developers, one of the standout features is the ability for VIC to expose containers directly on a network through the use of the container network option: vic-machine create –container-network. You can connect the containerVMs to any specific distributed port group or NSX logical switch, giving them their dedicated connection to the network.

The Benefits of Giving an App a Routable IP Address

This feature allows containerized applications to get their own routable IP and become first class citizens of your data center, providing the following benefits:

- No single point of failure: Now every container has its own dedicated network connection, so even if the VCH endpoint VM fails, there's no outage for your applications.

- No network bandwidth sharing: Every container gets its own network interface and all the bandwidth it can provide is available to the application. Traffic does not route through the VCH endpoint VM via network address translation (NAT), and containers do not share the public IP of the VCH.
- No NAT conflicts: There's no need for port mapping anymore. Every container gets its own IP address. The container services are directly exposed on the network without NAT, so applications that once could not run on containers can now run by using VIC.
- No Port conflicts: Since every container gets its own IP, you can have multiple application containers that require an exclusive port running on the same VCH. This provides better utilization of your resources.

All of this is possible through the use of the Container Network option.

The Container Network Firewall

But wait, there's more: To give vSphere administrators even better management and control over the traffic that flows on container networks, VIC includes a container network firewall:

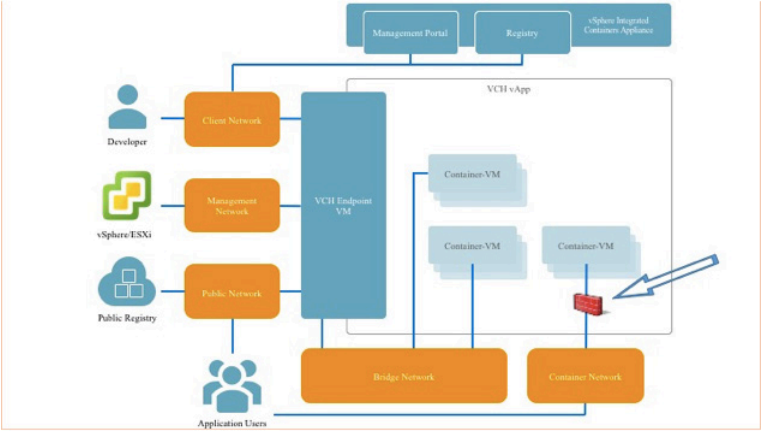

Figure 18: The container network firewall in vSphere Integrated Containers.

The container network firewall provides five distinct trust levels:

1. Closed: no traffic can come in or out of the container interface.

2. Open: all traffic is permitted.
3. Outbound: only outbound connections are permitted, which works well for containers that consume but do not provide services.
4. Published: only connections to published ports are permitted. When you create a container, you must specify which port will be permitted. (Default)
5. Peers: only containers on the same peer interface are permitted to communicate with each other. To establish peers, you need to provide an IP address range to the container network with the vic-machine create –container-network-ip-range option when you create a VCH.

The container firewall trust level is managed when you create a VCH:

```
vic-machine create --container-network-firewall "Port-
Group":[closed | open |outbound | published | peers]
```

In VIC version 1.2, the default trust level is set to Published. This means that you now have to explicitly identify which ports will be exposed with the -p option; example:

```
docker run -d -p 80 --network=external nginx
```

Running a container by using the -P option (e.g. docker run -d -P nginx) will not expose any service declared on the Dockerfile to the network, and your application will be unreachable from the outside.

Specifying the exposed port improves security and gives you more awareness of your environment and applications.

Now, if you still want to use the -P option (e.g. docker run -d -P nginx), you need to change the container network firewall trust level to Open:

```
vic-machine create --container-network "PortGroup" --con-
tainer-network-firewall "PortGroup":open
```

Power and Flexibility for Admins

As you can see, as a vSphere administrator, you get a lot of power and flexibility in your hands when configuring VCHs for your developers.

You can configure VCHs where no network traffic can come out of them, no matter what the developers try to do:

```
vic-machine create --container-network "PortGroup" --con-
tainer-network-firewall "PortGroup":closed
```

Or, you can configure VCHs where all traffic is permitted and you let the developer decide at the application level which ports are exposed and which are not:
vic-machine create --container-network “PortGroup” --container-network-firewall “PortGroup”:open

Or, you can configure VCHs where only outbound connections are permitted. This works well if you plan to host applications that consume but do not provide services:
```
vic-machine create --container-network “PortGroup” --container-network-firewall “PortGroup”:outbound
```

You can configure VCHs where only connections to published ports are permitted, letting the developers or DevOps control which ports are open for applications where you can’t change the Dockerfile. Think of all the new COTS applications delivered as Docker images:
```
vic-machine create --container-network “PortGroup” --container-network-firewall “PortGroup”:published
```

You can also configure VCHs where the containers can only communicate with each other. This is ideal for a set of microservices that need to talk with each other, but not with the external world. For example, a set of Spark jobs that compute some data and save the result to disk:
```
vic-machine create --container-network “PortGroup” --container-network-firewall “PortGroup”:peers
```

You should now have a better understanding of the benefits that the different networking options of VMware vSphere Integrated Containers, together with the Container Network Firewall feature, provide over traditional container host implementations, and how they make deploying containers on VIC even more secure. You should also know how to segregate different types of network traffic, make containers routable by exposing them directly on a network, and secure network connections by using the five distinct trust levels of the container network firewall.

Providing Persistent Storage for Legacy Applications

Linux containers have been great for stateless workloads. While stateful workloads can also run in containers, a limiting factor has been that most methods of providing storage for the state have been confined to the host serving the container. And if that host fails, the storage becomes inaccessible. Not so with vSphere Integrated Containers. It leverages vSphere’s advanced persistence capabilities to allow access to data even in the event of a host failure.

Let's dive into some of the storage concerns with standard container solutions and see how they are addressed in vSphere Integrated Containers.

Docker image layers

Container images are different from running containers. The images are static artifacts that are built and stored in Docker registries for use when running a new container. Images are just a set of files that make up the file system available to a running container.

Running containers are composed of layers of images applied in a stack. The underlying layers remain unaltered. While running, any changes to the file system will be persisted to an extra layer called the container layer. The container layer is removed when the container is removed.

Here is an example to illustrate what's going on in image layers. This Docker file builds on top of the alpine-3.6 image layer:

```
FROM alpine:3.6
RUN echo -e "#!/bin/sh\ndate\nsleep 2d\ndate" > /bin/our-ap-
plication
RUN chmod 755 /bin/our-application
CMD ["/bin/our-application"]
```

Building an image using this Docker file results in an image with several layers:

```
$ docker build -f Dockerfile.example-1 -t demo:0.1 .
Sending build context to Docker daemon  7.168kB
Step 1/4 : FROM alpine:3.6
 --- 76da55c8019d
Step 2/4 : RUN echo -e "#!/bin/sh\ndate\nsleep 2d\ndate" >/
bin/our-application
 --- Running in dfce6e80a2fb
 --- 9295df9995e6
Removing intermediate container dfce6e80a2fb
Step 3/4 : RUN chmod 755 /bin/our-application
 --- Running in cdc0e6d7ba27
 --- 1d5559a943d4
Removing intermediate container cdc0e6d7ba27
Step 4/4 : CMD /bin/our-application
 --- Running in d44e2734bef0
 --- 31af83e49686
Removing intermediate container d44e2734bef0
Successfully built 31af83e49686
```

```
Successfully tagged demo:0.1
```

The layers can be seen by running the docker history command:

```
$ docker history demo:0.1
IMAGE               CREATED             CREATED BY
SIZE
31af83e49686        2 minutes ago       /bin/sh -c #(nop)
CMD ["/bin/our-applicat...   0B
1d5559a943d4        2 minutes ago       /bin/sh -c chmod 755
/bin/our-application       29B
9295df9995e6        2 minutes ago       /bin/sh -c echo -e
"#!/bin/sh\ndate\nsleep...   29B
76da55c8019d        4 weeks ago         /bin/sh -c #(nop)
CMD ["/bin/sh"]               0B
            4 weeks ago          /bin/sh -c #(nop) ADD
file:4583e12bf5caec4...   3.97MB
```

The alpine layer is there at the bottom, and our additional commands have generated a few more layers that get stacked on top to be the image we want. The final image that has all the files we need is referenced by the ID 31af83e49686 or by the tag demo:0.1. Each of those layers should be stored in a registry, and can be reused by future images.

When we run the container, an additional container layer is created that allows modification of the file system by the running system. If no changes are made to the file system, this layer remains empty. Let's run the image as a container and modify its file system:

```
$ docker run -d --name demo --rm  demo:0.1
$ docker exec -it demo sh
/ # ls /
bin    dev    etc    home   lib    media  mnt    proc   root
run    sbin   srv    sys    tmp    usr    var
/ # date > /demo-state
/ # ls /
bin         dev         home        media       proc
run         srv         tmp         var
demo-state  etc         lib         mnt         root
sbin        sys         usr
```

As long as the container runs, the file demo-state will exist and have the same contents. Stopping and starting the container has no effect on the container layer, so demo-state will still exist.

If we stop and remove the container, the container layer will be removed as well as our hold on the demo-state file. Running a new instance of the container will have a new empty container layer.

For details on the structure of images, see https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

There is a distinction here that you should note: The container layer is ephemeral storage. It's around for as long as the container, and is lost when the container goes away. This is in contrast to requirements for data that needs to remain after the container is removed.

Why Aren't Containers Persistent?

The lack of persistence in the image layers is by design. By choosing to only allow ephemeral storage, we can ensure the application we put into a container image is always the application being run. Images are versioned so that we can be sure that two systems are running exactly the same code. Re-running the same image will always produce the same running conditions.

The immutability of the images results in better debugging, smoother deployments and the ability to quickly replace running applications that appear to be in a bad state.

Let's flip it around—if container images were able to change, how could you be sure running a specific image today and running it tomorrow would have the same results? How could you debug an image on my laptop and be sure you are seeing the same code that is having a problem in QA? If an application has persisted state in its local image, how do other instances of the application container get access to that data?

How Can You Save Data?

At some point, most of our applications need to leverage some data. How do we keep state between runs of an image? There are at least a few patterns:

- replication
- recreate data or replay transactions
- file system persistence

Replication

If you can design your application to replicate data to other containers and ensure at least one copy is always running, then you're using this pattern.

An example of this pattern is running a Cassandra database cluster, where replication enables the dynamic addition or removal of nodes. If you're running Cassandra in containers and being good about bootstrapping and removing nodes, then you could run a stable database cluster with normal basic docker run. The persistence is handled by storing data in the container layer. As long as enough containers are up, persistence is maintained.

Re-create or Replay Data on Loss

If you can design your application to be able to recreate any needed data, you're using this pattern.

An example of this might be a prime number finder tasked with finding a set of prime numbers in a broader range of numbers by counting up from the low end of the range and testing each number for primality. If the primes are stored for future use, but the data is lost for any reason, a new instance of the process can scan the same range and would find the same numbers that the original process found. In this case, the data is inherent to the requirements of the process, so the data can be recreated.

A more efficient variant of this process would store each prime number found and the last number tested in apache Kafka. Given a consistent initial range and the transaction log, you can quickly get back to a known state without retesting each number for primality, and continue processing from there.

Persistent File System

We can leverage an existing persistent file system that lives on the Docker host inside the container. This is a pattern most of us are familiar with, as it has been the way to handle data persistence since tape drives were invented.

Docker has two ways of handing a persistent file system in containers: bind mounts and volumes. Both of these expose a file system into the container from the running host. They are similar, but the bind mount is a bit more limited than using volumes.

Bind Mount

This is simply mounting a host file system file or directory into the container. This is not very different from mounting a CD-ROM onto a virtual machine (VM). The host path may look like /srv/dir-to-mount, and inside the container you may be able to access the directory at /mnt/dir-to-mount.

Bind mounting is used all the time in development, but should never be used in production. It ties the container to the specific host at runtime, and if the host is lost, so is the data. Volumes are the answer for production requirements.

Volumes

Volumes are the preferred way to use persistent storage in Docker.

This is slightly different from a simple bind mount. Here, Docker creates a directory that is the volume, and mounts it just like a bind mount. In contrast to bind mounts, Docker manages the lifecycle of this volume. By doing so, it provides the ability to use storage drivers that enable the backing storage to exist outside of the host running the container.

vSphere Integrated Containers leverages this to use vSphere storage types like vSAN, iSCSI and NFS to back the volume. Doing this means you can handle failures of any host running the container, and ensure access to the data in the volume can resume when the container is started on a different host.

Another example of leveraging the storage drivers of Docker volumes is shown in vSphere Docker Volume Service. This driver enables the use of vSphere-backed storage when using native Docker hosts, not vSphere Integrated Containers.

For deeper coverage on volumes, see Docker's volume document. Now, let's take a closer look at using volumes to persist data in vSphere Integrated Containers.

vSphere Integrated Containers Volumes

Command line use of volumes in vSphere Integrated Containers is the same as standard Docker, with the added benefit of the storage being backed by vSphere Storage.

In vSphere Integrated Containers, if you want to use volumes that are private to the container, you can use the iSCSI or vSAN storage in vSphere. If you have data that should be shared into more than one container, you can use an NFS backed datastore from vSphere.

When setting up a container host in vSphere Integrated Containers, you specify the datastores that will be available for use by any containers running against that host. These are specified using the --volume-store argument to vic-machine. These backing volume-stores can be set

or updated using vic-machine configure. Volumes added can only be removed by removing the container host, but that usually isn't a problem.

Here is an example showing the command that would create the container host and enable it to present volumes with various backing stores:

```
vic-machine ......
--volume-store vsanDatastore/volumes/my-vch-data:backed-up-encrypted
--volume-store iSCSI-nvme/volumes/my-vch-logs:default
--volume-store vsphere-nfs-datastore/volumes/my-vch-library:nfs-datastore
--volume-store 'nfs://10.118.68.164/mnt/nfs-vol?uid=0\&amp;gid=0:nfs-direct'
```

The first volume store is on a vSAN datastore and uses the label backed-up-encrypted so that a client can type docker volume create –opt VolumeStore=backed-up-encrypted myData to create a volume in that store. The second uses cheaper storage backed by a FreeNAS server mounted using iSCSI, and is used for storing log data. Note that it has the label "default," which means that any volume created without a volume store specified is created here. The third and fourth are for two types of NFS exports. The first being an NFS datastore presented by vSphere, and the other a standard NFS host directly (useful if you want to share data between containers).

Note regarding NFS gotcha: NFS mounts in container can be tricky. If you notice that you cannot read or write files to an NFS share in container, then you have probably hit this gotcha.Note the final volume store above has uid and gid arguments. There are two competing concerns. First, Docker will generally run as uid and gid 0, or as root. You can change that behavior by specifying a USER in the Dockerfile or on the command line. See Docker user command for details on how to set it. Second, NFS has many ways permissions based on uid and gid are applied to the mounted file system. You must ensure that the user of the running container matches the uid and gid permissions on the files exported by NFS. Finally, note that the syntax for native Docker NFS volumes and VIC NFS volumes is different, so if you are trying to apply this to native Docker, you'll want to start here.

Once you've installed the VCH, you'll notice that there are now empty folders created on the respective datastores ready for volume data:

```
vsanDatastore/volumes/my-vch-data/volumes
iSCSI-nvme/volumes/my-vch-logs/volumes
vsphere-nfs-datastore/volumes/my-vch-library/volumes
nfs://10.118.68.164/mnt/nfs-vol/volumes
```

Creating and Using Volumes

Let's go ahead and create volumes using the Docker client. Note the implied use of the default volume store in the second example.

```
$ docker volume create --opt VolumeStore=backed-up-encrypted
--opt Capacity=1G demo_data
$ docker volume create --opt Capacity=5G demo_logs
$ docker volume create --opt VolumeStore=nfs-datastore demo_
nfs_datastore
$ docker volume create --opt VolumeStore=nfs-direct demo_
nfs_direct
After volume creation, you'll see the following files were
created in the backing datastores:
vsanDatastore/volumes/my-vch-data/volumes/demo_data/demo_
data.vmdk
vsanDatastore/volumes/my-vch-data/volumes/demo_data/Image-
Metadata/DockerMetaData
iSCSI-nvme/volumes/my-vch-logs/volumes/demo_logs/demo_logs.
vmdk
iSCSI-nvme/volumes/my-vch-logs/volumes/demo_logs/ImageMeta-
data/DockerMetaData
vsphere-nfs-datastore/volumes/my-vch-library/volumes/demo_
nfs_datastore/demo_nfs_datastore.vmdk
vsphere-nfs-datastore/volumes/my-vch-library/volumes/demo_
nfs_datastore/ImageMetadata/DockerMetaData
nfs://10.118.68.164/mnt/nfs-vol/volumes/demo_nfs_direct
nfs://10.118.68.164/mnt/nfs-vol/volumes_metadata/demo_nfs_
direct/DockerMetaData
```

To show the most basic level of persistence, here we run a container that drops some data on each of the datastores and check that it exists from another container. In production, this could be a database workload hosted in a container and operating on the persistent external storage.

```
$ docker run -it --rm -v demo_data:/data -v demo_logs:/logs -v demo_
nfs_datastore:/library -v demo_nfs_direct:/shared busybox sh

# echo "some data" > /data/some-data ;
# echo "some logs" > /logs/some-logs ;
# echo "some library" > /library/some-lib;
# echo "some shared" > /shared/some-shared ;
# exit
$
$ docker run -it --rm -v demo_data:/data -v demo_logs:/logs
-v demo_nfs_datastore:/library -v demo_nfs_direct:/shared
alpine  sh
```

```
# cat /data/some-data /logs/some-logs /library/some-lib /
shared/some-shared
# exit
```

Right now, only native NFS volumes are allowed to share data between more than one container. Here is an example of sharing some storage between containers using native NFS.

Open two terminals. In the first run this command to start nginx:
$ docker run --name nginx -v demo_nfs_direct:/usr/share/nginx/html:ro -p 80:80 -d nginx

In the same terminal query, the nginx server you just created. Replace 192.168.100.159 with the ip address of the container running nginx:

```
$ curl 192.168.100.159
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

In the second terminal, run this command to simultaneously mount the shared filesystem and edit a file:

```
$ docker run -it --rm -v demo_nfs_direct:/shared busybox sh
# echo "hello from $(date)" > /shared/index.html
Now that you've overwritten the shared index.html, go back
to the first terminal and rerun the curl command. You should
see something like this:
hello from Mon Oct 16 21:33:03 UTC 2017
```

As a final note, if you have a stateful process that can handle restart, VMware HA will enable restarting the container on a new ESXi host if the original ESXi host fails. If your process can't implement a replay or replication pattern to recover state on failure, then VMware Fault Tolerance enables transparent continuation of processing during an ESXi host failure. In this case the container VM continues running on the new ESXi host as though there were no failure of the original host. We'll see if we can make a blog entry demonstrating the Fault Tolerance feature.

Here is an example of VMware HA helping a container resume running on a new host after failure of the initial ESXi host. This is the picture before failure:

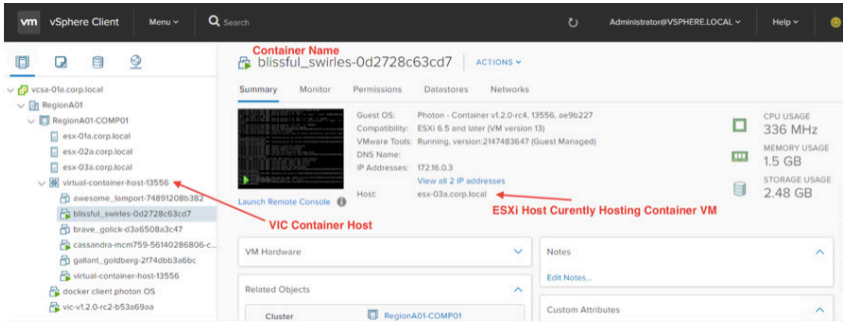

Figure 19: VMware HA helping a container resume running on a new host.

And after causing an ESXi host failure, the container is moved to and started on a different ESXi host:

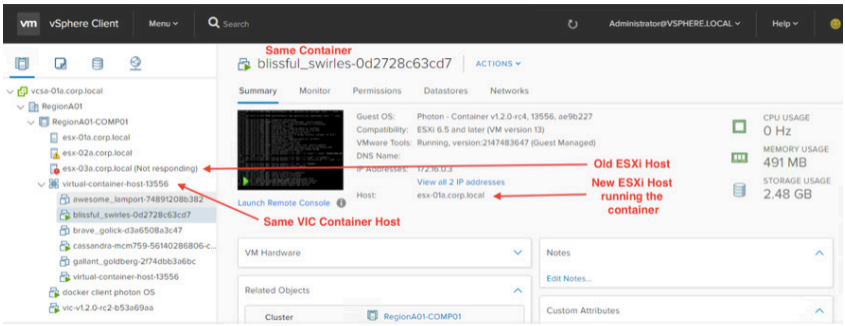

Figure 20: The container is moved to and started on a new ESXi host.

So, there you have it: vSphere Integrated Containers can provide resilient storage and cope with host failures. It's not mandatory during development, but definitely a boon in the production landscape.

Setting Up a Developer Sandbox

vSphere Integrated Containers (VIC) includes the concept of the native Docker Container Host (DCH). It is a built-in Docker image containing a full-fledged Docker engine that runs using VIC. DCH is packaged as a container and can be instantiated on VIC like any regular container.

The DCH image is distributed through Docker Hub and, as part of the VIC product distribution, in the registry. All the official DCH images maintained by VMware are based on Project Photon OS, an open source Linux operating system optimized for hosting containers and running cloud-native applications. The source, Dockerfiles and documentation are available at github.com/vmware/vic-product.

DCH is well-suited for development use cases. Here are some examples:

- As part of a CI/CD pipeline, VIC can be used to enhance end-to-end dev-build-push-deploy workflows. VIC with DCH can be used as a (self-service) private cloud for CI/CD by enabling the easy deployment and tear down of Docker hosts.
- VIC and DCH allow you to treat Docker Hosts as ephemeral compute. This has the benefit of eliminating snowflakes (individually managed Docker Hosts), which reduces Operating System OpEx costs. For example, as part of a CI pipeline, you could instantiate ephemeral Docker Hosts that exist only for the purpose of building and pushing images, and only for the time it takes to complete that task.
- An example of how VIC can be used to deploy Jenkins is given here.

This section demonstrates how flexible this DCH abstraction is. The section walks you through how a developer can leverage VIC 1.2 and DCH to easily instantiate a Docker swarm using only well-known native Docker commands. The section also shows how easy it is to create a complete cluster of ephemeral compute using DCH.

Scripting the Deployment of Swarm Manager and Worker Nodes

To illustrate this, look at the following script. This is a simple shell script that deploys a Docker swarm manager node and then creates and joins a user-defined number of worker nodes to the swarm.

```
    #!/bin/bash
    ## USER-DEFINED VARIABLES
    # Number of swarm workers desired
    NUM_WORKERS=3
    # name of routable (external) network
    # this needs to be defined on your VCH using the '--con-
tainer-network' option
    # use 'docker network ls' to list available external
```

```
networks
    CONTAINER_NET=routable
    # Docker Container Host (DCH) image to use
    # see https://hub.docker.com/r/vmware/dch-photon/tags/
for list of available Docker Engine versions
    DCH_IMAGE="vmware/dch-photon:17.06"
    ## NO NEED TO MODIFY BEYOND THIS POINT
    # pull the image
    docker pull $DCH_IMAGE

    # create a docker volume for the master image cache
    docker volume create --opt Capacity=10GB --name regis-
trycache
    # create and run the master instance
    docker run -d -v registrycache:/var/lib/docker \
    --net $CONTAINER_NET \
    -name manager1 -hostname=manager1 \
    $DCH_IMAGE
    # get the master IP
    SWARM_MASTER=$(docker inspect -f '{{range .NetworkSet-
tings.Networks}}{{.IPAddress}}{{end}}' manager1)
    # create the new swarm on the master
    docker -H $SWARM_MASTER swarm init
    # get the join token
    SWARM_TOKEN=$(docker -H $SWARM_MASTER swarm join-token
-q worker)
    sleep 10

    # run $NUM_WORKERS workers and use $SWARM_TOKEN to join
the swarm
    for i in $(seq "${NUM_WORKERS}"); do

    # create docker volumes for each worker to be used as
image cache
    docker volume create  --opt Capacity=10GB --name work-
er-vol${i}
    # run new worker container
    docker run -d -v worker-vol${i}:/var/lib/docker \
    --net $CONTAINER_NET \
    --name worker${i} --hostname=worker${i}  \
    $DCH_IMAGE
    # wait for daemon to start
    sleep 10

    # join worker to the swarm
```

```
    for w in $(docker inspect -f '{{range .NetworkSettings.
Networks}}{{.IPAddress}}{{end}}' worker${i}); do
    docker -H $w:2375 swarm join --token ${SWARM_TOKEN}
${SWARM_MASTER}:2377
    done

    done

    # display swarm cluster information
    printf "\nLocal Swarm Cluster\
n=========================\n"

    docker -H $SWARM_MASTER node ls

    printf "=========================\nMaster available at
DOCKER_HOST=$SWARM_MASTER:2375\n\n"
```

Let's break this down to better understand what this script does.

User-defined Variables

```
    ## USER-DEFINED VARIABLES
    # Number of swarm workers desired
    NUM_WORKERS=3
    # name of routable (external) network
    # this needs to be defined on your VCH using the '--con-
tainer-network' option
    # use 'docker network ls' to list available external
networks
    CONTAINER_NET=routable
    # Docker Container Host (DCH) image to use
    # see https://hub.docker.com/r/vmware/dch-photon/tags/
for list of available Docker Engine versions
    DCH_IMAGE="vmware/dch-photon:17.06"
```

As a user of the script, this is the only section you need to modify.
NUM_WORKERS – this is the number of worker nodes that will be added to the swarm, in addition to the manager node.
CONTAINER_NET – this is the network to be used by our Docker Container Hosts. Here we leverage the ability of vSphere Integrated Containers to connect containers directly to vSphere Port Groups rather than through the Container Host. This will allow for easier interaction with our swarm.
DCH_IMAGE – here you can specify a different version of the Docker engine by modifying the tag (e.g. 'vmware/dch-photon:1.13'). You can see the list of available tags/versions here.

The script will use these parameters to pull the images, instantiate the swarm manager, initiate the swarm and instantiate and join the user-defined number of worker nodes.

Creating the master, initiating the swarm and getting the join token

```
    # create a docker volume for the master image cache
    docker volume create --opt Capacity=10GB --name registrycache
    # create and run the master instance
    docker run -d -v registrycache:/var/lib/docker \
    --net $CONTAINER_NET \
    --name manager1 --hostname=manager1 \
    $DCH_IMAGE
    # get the master IP
    SWARM_MASTER=$(docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' manager1)
    # create the new swarm on the master
    docker -H $SWARM_MASTER swarm init
```

This is where we begin to see the DCH magic in action. Specifically, look at the following command:

```
docker run -d -v registrycache:/var/lib/docker \
--net $CONTAINER_NET \
--name manager1 --hostname=manager1 \
$DCH_IMAGE
```

This starts up a full-fledged docker engine (running version 17.06) instantiated as a container VM, using only a simple docker run command. As you can see, DCH makes it very easy for developers to start up docker engines on-demand, thus creating new self-service possibilities.

Once the manager is created, we initialize the swarm (docker -H $SWARM_MASTER swarm init), and then we grab the join token (docker -H $SWARM_MASTER swarm join-token -q worker) that will allow us to join the worker nodes to the swarm in the next step.

Creating the workers and adding them to the swarm

```
    # run $NUM_WORKERS workers and use $SWARM_TOKEN to join the swarm
    for i in $(seq "${NUM_WORKERS}"); do
```

```
    # create docker volumes for each worker to be used as
image cache
    docker volume create  --opt Capacity=10GB --name work-
er-vol${i}
    # run new worker container
    docker run -d -v worker-vol${i}:/var/lib/docker \
    --net $CONTAINER_NET \
    --name worker${i} --hostname=worker${i}  \
    $DCH_IMAGE
    # wait for daemon to start
    sleep 10

    # join worker to the swarm
    for w in $(docker inspect -f '{{range .NetworkSettings.
Networks}}{{.IPAddress}}{{end}}' worker${i}); do
    docker -H $w:2375 swarm join &ndash&ndashtoken ${SWARM_
TOKEN} ${SWARM_MASTER}:2377
    done

    done
```

This simple `for' loop repeats NUM_WORKERS times. For each iteration, it:

- creates a volume to be used as the image cache for the worker
- instantiates a worker using the vmware/dch-photon:17.06 image
- joins the worker to the swarm using the join token (SWARM_TOKEN) we fetched in the previous step

Once again, because we are using DCH with VIC, we see that it requires only a very simple docker run command to create and run our Docker Hosts that will be used as the swarm worker nodes.

Running The Script And Result

Before running the script, you need to point your Docker client to a VIC Virtual Container Host endpoint. This is done by setting DOCKER_HOST=<endpoint-ip>:<port>. This script requires a Virtual Container Host endpoint—a requirement to run the DCH image, which enables all of the above automation. You can find this information in the vSphere Client portlet:

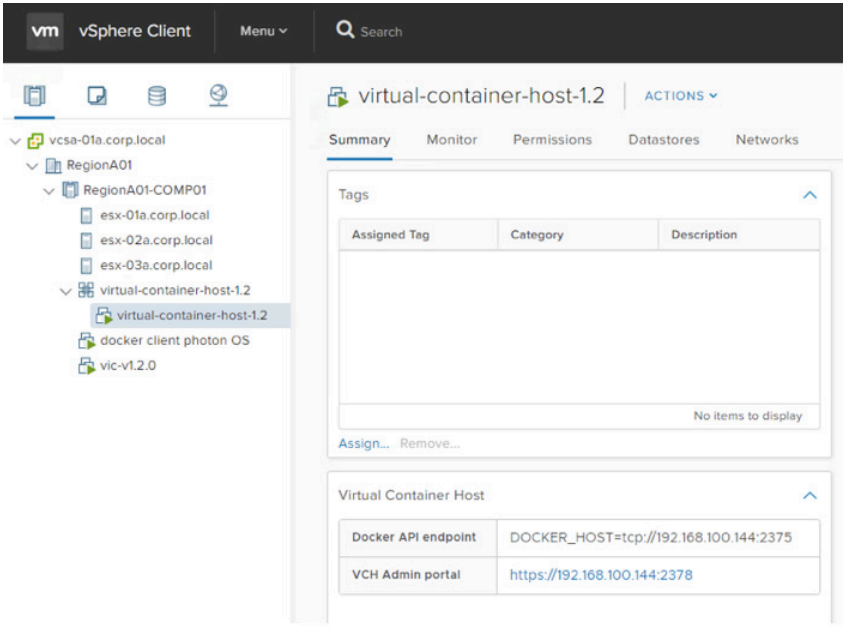

Figure 21: The vSphere Client.

You can use the following commands (replace with your own endpoint IP address and make sure the script is in the current directory):

```
export DOCKER_HOST=192.168.100.144:2375
./dch-swarm.sh
```

Upon completion, the script prints information about the newly created swarm:

```
Local Swarm Cluster
========================
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
2oj34vs3lknl70gjjrba46qy9    worker1   Ready   Active
i9yszvipobq1vzjrk96cwgnod *  manager1  Ready   Active        Leader
j7jdcj3wwh5nijgmsfjbxy8bp    worker3   Ready   Active
pkm3avxzsva9xfj6j0a0n7a6t    worker2   Ready   Active
========================
Master available at DOCKER_HOST=192.168.100.167:2375

root@docker-client [ ~/src/dch-swarm ]#
```

Figure 20: Information about a newly created Swarm cluster.

We can also see our newly created containerVMs hosting the swarm manager and the swarm workers in vSphere Client:

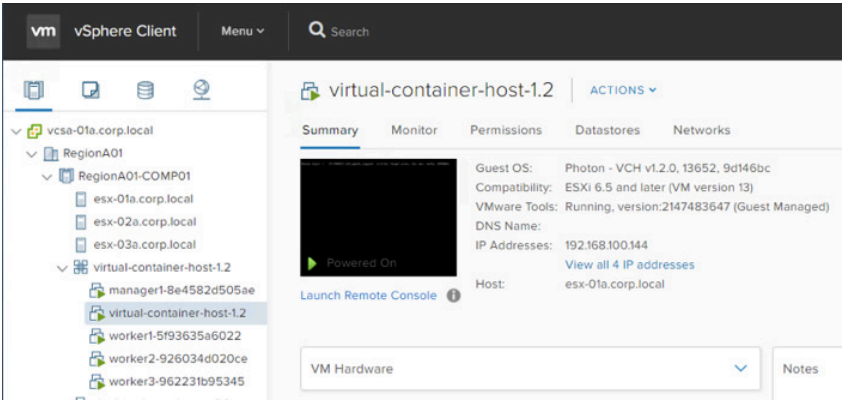

Figure 22: Container VMs holding the swarm manager and workers.

Finally, we can test the swarm by running the Docker Example Voting App. We deploy the app against our swarm by using the docker stack deploy command:

```
root@docker-client [ ~/src/example-voting-app ]#  export DOCKER_HOST=192.168.100.167:2375
root@docker-client [ ~/src/example-voting-app ]# docker stack deploy --compose-file docker-stack.yml vote
Creating network vote_frontend
Creating network vote_backend
Creating service vote_redis
Creating service vote_db
Creating service vote_vote
Creating service vote_result
Creating service vote_worker
root@docker-client [ ~/src/example-voting-app ]# docker service ls
ID            NAME         MODE        REPLICAS  IMAGE
f54v8hvj8o9r  vote_worker  replicated  1/1       dockersamples/examplevotingapp_worker
m1aay56zioox  vote_db      replicated  1/1       postgres:9.4
mfxysbldlmwn  vote_vote    replicated  2/2       dockersamples/examplevotingapp_vote:before
p7nufrierjok  vote_redis   replicated  1/1       redis:alpine
z7qgavch6p5t  vote_result  replicated  1/1       dockersamples/examplevotingapp_result:before
root@docker-client [ ~/src/example-voting-app ]#
```

Figure 23: Running the docker stack deploy command.

We can see above that all of the required services were started. Testing the application in the browser, we can see it is indeed running and functional:

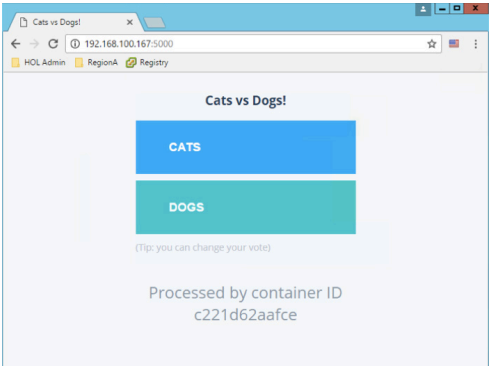

Figure 24: The example app is running.

Conclusion

You should now have a better understanding of how the newly introduced VIC DCH feature helps address developer use cases. We have shown how easy it is to automate the deployment of Docker hosts using DCH. VIC provides end-users and developers with a Docker dial tone and a very flexible consumption model on top of vSphere, while DCH enables a new level of self-service.

If you're interested in getting more hands-on knowledge around VIC, check out our tutorials on GitHub: https://github.com/vmware/vic-product/tree/master/tutorials

Deploying Jenkins by Using VIC

Containers have fundamentally changed how we consume software. They've changed how we develop and package software. Containers have also changed how we interact with infrastructure.

It has never been easier to deploy a service, an application, a database or a cluster than with today's container technologies. Consider the following:

- The immutability and portability of container images means that workloads deploy predictably in multiple locations and scale up and down with ease.
- The increasing sophistication of container registries incorporate security capabilities and access control that simply weren't available previously.

- The way that containers force developers to think about how state is managed, in terms of persistence, scope and integrity is leading to more flexible application architectures.

VMware vSphere Integrated Containers has brought all of these benefits directly to VMware vSphere by giving you the ability to manage and consume vSphere infrastructure using an prescribed container consumption model.

What's even better about vSphere Integrated Containers is that it goes where many containers fear to tread. Take a large database for example. Many people would tell you not to run a database in a container. Why? Because you need data integrity, strong runtime isolation, and good network throughput. With a regular Linux container, that would mean deploying a dedicated Linux host, securing and patching it, configuring it with a volume plugin that works with a storage LUN of some kind, ensuring that no other containers run in that host, and making sure to configure the container with host networking. vSphere Integrated Containers requires no such configuration. It will deploy a MySQL or MSSQL container image out-of-the-box direct to vSphere as a strongly isolated virtual machine that has encrypted, replicated persistent storage on VMware vSAN, with its own vNIC and connected directly to an NSX logical switch.

Deploying workloads to vSphere has never been so easy. Jenkins is a great example of a long-running stateful application that plays to vSphere Integrated Containers's strengths.

This section discusses the best practices around deploying Jenkins by using vSphere Integrated Containers. The section covers how to maintain the persistent state of Jenkins, how to configure security around image management and access control, how to ensure that it has the resources it needs, and how vSphere High Availability (HA) can make the master node highly available.

Few would argue with the contention that containers make software provisioning easier. As such, with vSphere Integrated Containers, it's never been easier to provision software to vSphere. This is just as true of Jenkins as any other application. However, with power comes responsibility and if we want to deploy Jenkins, there's a few critical factors we need to consider:

- What are the software artifacts in the Jenkins image we want to deploy?
- Do we trust the provenance of those artifacts?
- Do those software artifacts contain known vulnerabilities?

- What data needs to be persisted and what data should be ephemeral?
- What resource limits should we put around the container?
- Do we want the container to be highly available and if so, how?

Security of Software Artifacts

Deploying an image from a public registry without knowing its contents or provenance is risky. To help you address these security concerns, vSphere Integrated Containers comes with its own registry.

As a cloud admin, you can choose to build your own container images using your own Dockerfiles, or you can start from a public image and further modify it to your needs. As you'll see from DockerHub, you can choose from a Debian base or an Alpine base. The Alpine base is half the download size and has less than half the packages, so that makes it attractive, although there may be compliance considerations involved in the decision.

As an example, let's start by running the following commands to pull the jenkins/jenkins:lts-alpine image from DockerHub, push it to a registry, and scan it for vulnerabilities.

```
docker pull jenkins/jenkins:lts-alpine
docker tag jenkins/jenkins:lts-alpine vicregistry.myfirm.com/
myproject/jenkins:lts-alpine
docker push vicregistry.myfirm.com/myproject/jenkins:lts-al-
pine
```

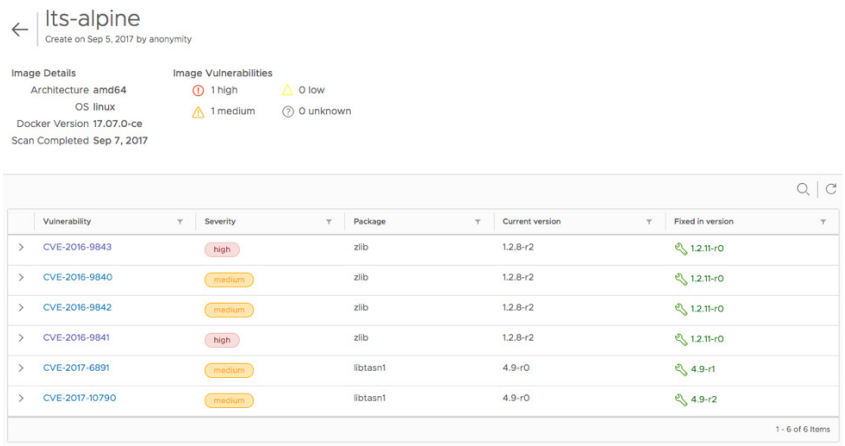

Figure 25: The vulnerabilities of an image.

At the time these commands were run, vSphere Integrated Containers registry identified that the zlib package contains two high-level vulnerabilities. (Your results might be different when you run the command.) It provides a link to the CVE database, which describes the issue. It also shows that there are updated versions of zlib in which the issue is fixed, so we can use that information to create a Dockerfile that defines a new image with updated packages.

```
cat Dockerfile
FROM jenkins/jenkins:lts-alpine
USER root
RUN apk update && apk upgrade
USER jenkins

docker build -t vicregistry.myfirm.com/myproject/jenkins:lts-alpine-upgrade .

docker push vicregistry.myfirm.com/myproject/jenkins:lts-alpine-upgrade
```

Once the image is pushed, the vSphere Integrated Containers registry shows that it is 100 percent green—no vulnerabilities.

Figure 26: A pushed image with no vulnerabilities..

As a cloud admin, you can choose to limit the vulnerability level images can be deployed with. You can also restrict the images deployed to an endpoint to only ones that have been signed by a service such as Notary.

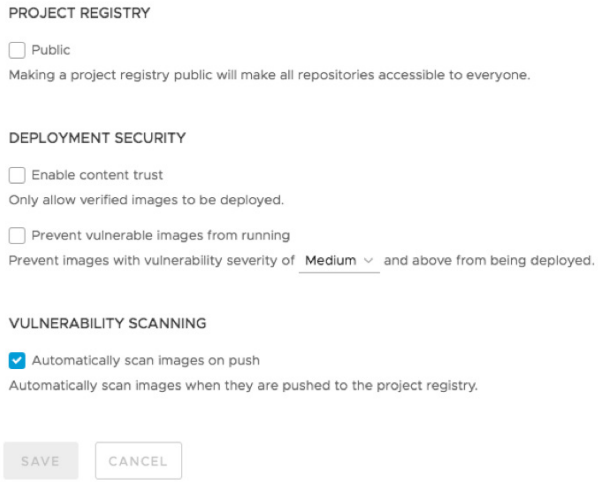

Figure 27: Security thresholds for images.

Data Persistence

The Jenkins container is configured in such a way that all of the persistent state is stored in one location: /var/jenkins_home. This means that you can safely start, stop or even upgrade the Jenkins master container and it will always come back up with all of the previous data, assuming you specified a named volume.

This data includes job definitions, credentials, logs, plugins—important data that should not only be persistent but should also have high integrity. This is not data you want to have to recreate. vSphere Integrated Containers makes it easy to store persistent data with these characteristics by mapping a container volume to a persistent disk on a vSphere datastore. The volume can then benefit from the security capabilities of the datastore, such as encryption, and replication on a vSphere vSAN.

Resource Limits and HA

vSphere Integrated Containers makes it easy to specify how much resource a container should consume when deployed by allowing for the specification of vCPUs and a memory limit.

A vSphere Integrated Containers container has exclusive access to its own guest buffer cache, so there's no resource competition from other containers.

If the vSphere cluster has HA enabled, then if an ESXi host goes down, the endpoint VM and the containers will be automatically restarted on other hosts.

Deploying Jenkins and Access Control

Now that we've thought about all of the implications, we can go ahead and deploy Jenkins. This can be done either with the vSphere Integrated Containers Management UI or by using an ordinary Docker command-line client.

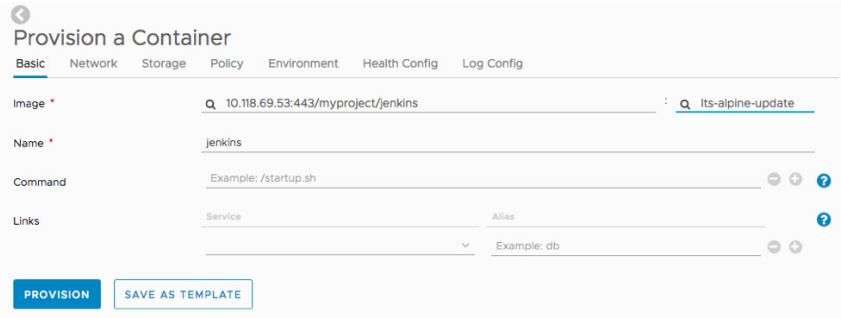

Figure 28: Provisioning a container.

Regardless of how it's done, it should only be possible to authenticate with the vSphere Integrated Containers endpoint with the appropriate credentials. As a cloud admin, the Management UI gives you control over who has access to those credentials, thereby limiting who has access to certain deployment endpoints and ensuring that credentials are not leaked.

In addition, vSphere Integrated Containers integrates with the vSphere Platform Services Controller, which means that identities in the vSphere Integrated Containers Management UI can be vSphere identities, but with additional roles and responsibilities.

The Docker command-line that could be used to deploy Jenkins might look like this:

```
docker volume create -opt VolumeStore=encryped -opt Capaci-
ty=5G my-named-volume

docker run -d -name jenkins-master -cpuset-cpus 2 -m 4g
-p 8888:8080 -e TINI_SUBREAPER= -v my-named-volume:/var/
jenkins_home vicregistry.myfirm.com/myproject/jenkins:lts-al-
pine-upgrade

docker logs jenkins-master
```

Note that a named volume on the desired datastore should be created first and then mapped to the appropriate mount point. Setting the environment variable TINI_SUBREAPER to null ensures that the Tini init process functions correctly, given that it won't run as PID 1 in a vSphere Integrated Containers container. When you start Jenkins, you need an initial admin password that's generated to the logs, so the docker logs command will show you that password.

If you deploy the same container via the Management UI, you can create a template that persists the configuration so that you can re-use it for future deployments. Viewing the container once it's deployed allows you to see statistics and logs for the container.

Once Jenkins is up and running, you can access it at http://vch-end-point-address:8888. It will ask for the password from the logs and then ask you to create an Admin user. You can go ahead and take the default plugins and once they've finished installing, you should see the Jenkins dashboard ready for configuration.

Optimizing Cloud-Native Apps with PCF and Developer-Ready Infrastructure from VMware

Developer-ready infrastructure lays the foundation for agile application development. A data center modernized with secure, software-driven compute, storage, and networking can rapidly fulfill the needs of developers seeking to build and deploy modern applications. When developer-ready infrastructure reduces manual IT infrastructure processes and provides the operational tooling required to run containerized workloads at scale, developers can focus their energy on delivering robust cloud-native applications architected with microservices.

By scaling to meet demand and maintaining high availability for applications, Pivotal Cloud Foundry addresses the needs of modern agile development techniques and application architectures. But if the under-

lying infrastructure is unable to match the platform's resource demands, performance can be impaired. Developer-ready infrastructure from VMware supplies three key services that dynamically optimize resources for Pivotal Cloud Foundry:

- Scalability
- Availability
- Security

This section explains how developer-ready infrastructure from VMware helps Pivotal Cloud Foundry run in a demanding production environment. In particular, it examines how developer-ready infrastructure from VMware works with Pivotal Cloud Foundry to address issues of scale, availability, and security.

THE BENEFITS OF DEVELOPER-READY INFRASTRUCTURE

Developer-ready infrastructure helps modernize application development, yielding benefits that ultimately bolster your competitive advantage

- Provide a consistent, cloud-independent networking and security layer
- Eliminate bottlenecks that hinder the provisioning of IT resources for developers
- Reduce manual infrastructure processes
- Improve developer agility and productivity
- Shorten software's time to market

How Pivotal Cloud Foundry Works

Pivotal Cloud Foundry (PCF) is a logical collection of services that provide a platform upon which developers can run applications and microservices in a stable, consistent, and fault-tolerant manner. With native support for Java, Node, Golang, .Net, and many others, the platform supports applications written in multiple languages. PCF automatically detects your applications' required runtime when applications are published to it, giving a developers a range of options for writing code with the language of their choice.

Applications are usually published on PCF through a continuous integration and continuous development pipeline. Using a CI/DI pipeline helps developers quickly build, test, and deploy an application in a reliable, repeatable, and automated way. With the PCF CI/CD model, developers

first check their source code into a source repository like Git. A pipeline automation engine, such as Jenkins, then triggers the commit of the updated source code to Git. The result is an application artifact. For example, a pipeline would produce a JAR or WAR file as an artifact for a Java application. The pipeline then pushes the artifacts to PCF to be instantiated as containers.

When an application is pushed to PCF, the artifact is staged, creating an image that can be run as a container in PCF's container execution environment called Diego. In the staging process, PCF first identifies the artifact type in order to match the appropriate "buildpack" to use in staging the application image. A buildpack is a library of application runtime components. For example, the Java buildpack includes components such as the Spring Boot framework and the TC server runtime to run the Java application. In this way, containers streamline the process of developing and staging an application.

THE BENEFITS OF MICROSERVICES INCREASE MODULARITY

- Make app easier to develop and test
- Parallelize development: A team can develop and deploy a service independently of other teams working on other services
- Support continuous code refactoring to heighten the benefits of microservices over time
- Drive a model of continuous integration and delivery
- Improve scalability
- Simplify component upgrades

Running Application Instances

The staging process composes an application image by combining a read-only root file system with the buildpack and the application artifacts. This image, packaged as a tgz, is called a 'droplet.' PCF can then use its Diego execution environment to run the droplet images as application instances (AIs). Diego schedules the running of all application instances and maintains their availability across the platform, which reduces the workload of IT teams.

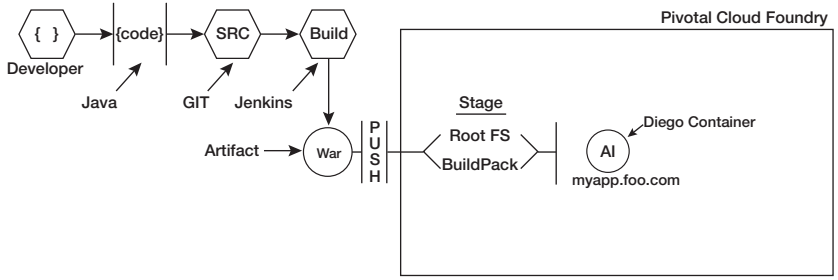

Figure 29: The application pipeline and PCF.

Running an App by Using a Docker Image

PCF is also capable of running an application from a Docker image. Docker images are uploaded to PCF pre-composed, so the push process doesn't stage in the same way as a Java application, but the end result is the same: an application instance managed by PCF and Diego.

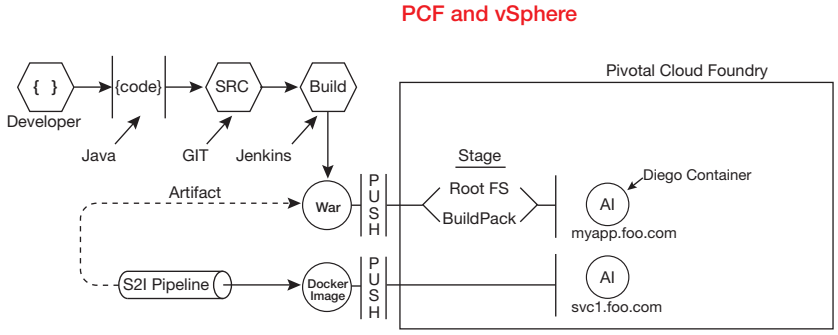

Figure 30: PCF with a Docker pipeline.

THE BENEFITS OF CONTAINERS FOR DEVELOPERS

Developers like working with containers because they make life easier, development more engaging, and work more productive.

- Portability: Containers let developers choose how and where to deploy an app.
- Speed: Containers expedite workflows like testing and speed up iterations.
- CI/DI Pipeline: Containers support continuous integration and deployment.
- Flexibility: Developers can code on their laptops when and where they want with the tools they like.

Reliable Service at Scale

In addition to running containers, PCF offers application routing services, making it possible to bind multiple routes and domain URLs to any application. PCF also ensures that application instances maintain a minimum level of availability, offering protection from failures in the physical infrastructure hosting PCF, as well as easily scaling application instances dynamically to meet changes in demand.

PCF does this by scheduling multiple AIs per application and running them in multiple PCF availability zones. These zones, typically created in sets of three, maintain an application's uptime if a fault occurs in the infrastructure backing a zone.

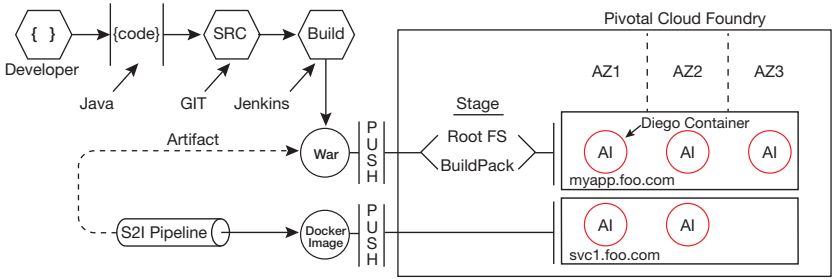

Figure 31: PCF application instances and availability zones.

Delivering IaaS with VMware Solutions

Today, the infrastructure underlying Pivotal Cloud Foundry is typically delivered through an Infrastructure as a Service (IaaS) solution like VMware vSphere® and VMware vCenter®.

PCF interacts with the IaaS through BOSH, an open source tool for managing the lifecycle of distributed systems. Pivotal Cloud Foundry deploys BOSH as a virtual machine called the Ops Manager Director. BOSH creates VM instances and assigns one or more jobs to each VM instance. BOSH jobs provide a VM instance with desired service release components of PCF; for example, a job called diego_cell contains all of the release components required to stage and start containers in PCF. BOSH will then ensure availability of all PCF services by deploying VM instances across availability zones and ensuring jobs are assigned to them across availability zones as well.

BOSH communicates with the IaaS through a cloud provider interface, or CPI, to create, update, and remove VM instances. BOSH also uses an agent communication path with the guest operating system of the VM instance to automatically push configured jobs and collect service health. If health is determined to not be in a desired state, BOSH can resurrect or rebuild the instance and re-apply its jobs. This process ensures that PCF services remain available even during faults in a given availability zone.

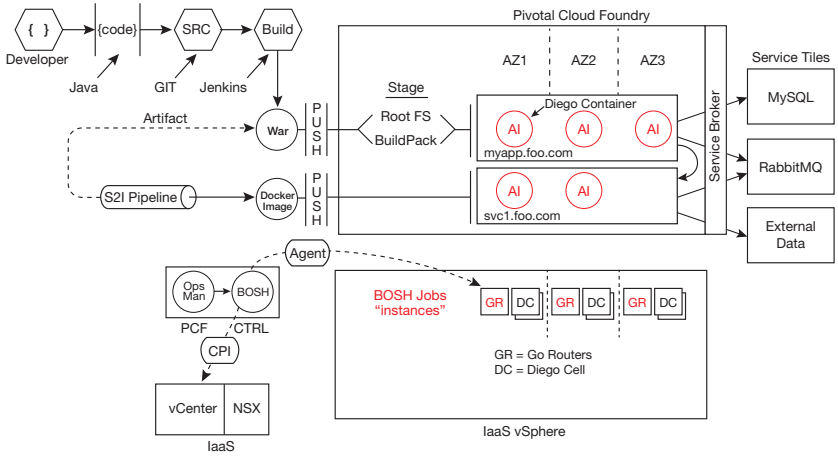

Figure 32: PCF and vSphere.

Describing Deployments with Manifests

Configuration and interaction with BOSH is a potentially cumbersome process that sometimes requires wrangling large YAML text files known as manifests. Manifests describe how BOSH should deploy and scale PCF and PCF services on a given IaaS.

Pivotal provides a tool called Operations Manager, which also communicates with the IaaS through the CPI to create and manage BOSH itself. Operations Manager offers an API as well as a graphical interface for a platform operator to configure how PCF is deployed. Additionally it provides the platform operator a tool to scale, apply updates, and patch PCF. This architecture improves the maintainability of the platform.

Self-Service Access for Rapid Application Development

Operations Manager allows a platform operator to deploy and manage application services as part of their PCF deployment. PCF services can provide developers policy driven, self-service access to components their applications require through a "marketplace." These services, such as MySQL, RabbitMQ, and Redis, are instantiated by Pivotal Ops Manager and BOSH across the availability zones.

PCF can also allow policy-driven access to services external to those it manages, including mainframe data and other vendor-external information and services. This is enabled by the PCF service broker layer, which allows the platform to bind services and credentials to any application.

Scalability

While there are many ways in which Pivotal Cloud Foundry is designed to scale, one of the most important is in scaling application instances. These run on VM instances in PCF called Diego Cells.

Diego Cell-instance VMs are pre-created by BOSH for application instances (AIs) to run on, and so when application instances start to scale horizontally or become larger, they may require more Diego Cell VM instances. It's essential that the IaaS hosting Pivotal Cloud Foundry can also scale its resources to allow BOSH to quickly add additional PCF VM instances and associated jobs.

Supporting Multiple PCF Deployments on vSphere Infrastructure

In a developer-ready infrastructure, that construct to scale resources is the vSphere cluster. Defined within each cluster are one or more resource pools, each of which is typically aligned with a PCF availability zone. One or more availability zones align to a deployment of PCF. Aligning availability zones to resource pools allows for multiple deployments of PCF across a common infrastructure. For example, three vSphere clusters, each with two resource pools, can support two PCF foundations in a highly available architecture of 3 x PCF availability zones. Resource pools can also allow the platform operator to enable resource reservations and limits when sharing multiple PCF deployments on common clusters. As a result, vSphere can dynamically add or expand resources, hosts, and storage without affecting PCF or its applications.

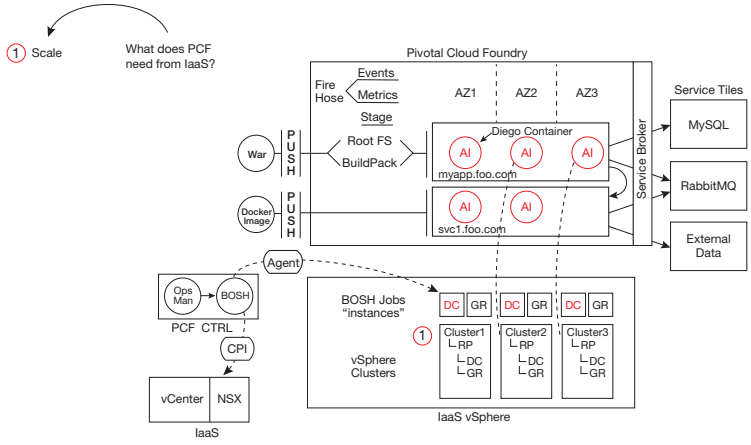

Figure 33: PCF and vSphere for scalability..

DYNAMICALLY SCALING RESOURCES FOR APPLICATIONS WITH VSPHERE

- Create multiple deployments of Pivotal Cloud Foundry across common infrastructure
- Quickly and dynamically scale resources to add virtual machines and run more application instances
- Reserve resources and set consumption limits
- Add storage without affecting running applications

Availability

Another key area in which vSphere supports Pivotal Cloud Foundry is availability. PCF interacts with the IaaS through BOSH, which can detect VM instance health and make sure that VM instances and their jobs stay running. One way it does this it by automatic repairs and rebuilds to all PCF VM instances when they are found to be unhealthy or unresponsive.

vSphere HA adds a complementary service here, helping reduce down-time in case of a physical failure or in case of undesired VM guest faults. The two technologies need to be scheduled appropriately so that they don't step over each other, but each works with the other to ensure that PCF instances and their jobs are always available.

Availability Zones and Load Balancing

PCF also achieves availability through its availability zone constructs. As noted earlier, application instances are distributed across Diego Cell VM instances in multiple PCF availability zones. Another core PCF job called a "router" is also required to route requests to the application instances. Router jobs, also known as "GoRouters," run in VM instances that will load balance application requests to AIs running on Diego Cell VM instances, across availability zones. This means router instances must also be highly available and deployed across availability zones to allow requests to reach their desired applications.

Other issues can affect availability. These issues can exist within both PCF and the infrastructure hosting it. Platform operators must implement effective monitoring of both PCF and the infrastructure hosting it. This is critical to maintaining availability and scale.

Tracking Performance Indicators and Events with vRealize Operations

Monitoring of key performance indicators (KPIs) and events is accomplished via a PCF service called the "firehose." The firehose endpoint will stream events and metrics for all PCF components, BOSH VM instances, the jobs running on those instances, and all application instances (AIs). This is essential because while BOSH makes deployment simple, PCF itself is a complex distributed system that requires careful day-two operations and management. Tapping into the firehose stream, VMware vRealize® Operations™, a component of the VMware vRealize® Suite, can help solve this platform operations problem.

vRealize Operations and VMware vRealize® Log Insight™ ingest PCF KPIs and events as well as IaaS metrics and events from vCenter. Data, such as

application instance growth patterns and the speeds at which infrastructure resources are being consumed by developer tenants, allows vRealize Operations to track if and when the underlying infrastructure capacity will be exceeded. vRealize Operations also visualizes the complete deployment of PCF in a set of dashboards detailing PCF key performance indicators, and can alert you when there are unhealthy KPIs. vRealize allows an operator to track and act upon significant log events that may indicate PCF service and application failures. vRealize Operations and vRealize Log Insight provide a comprehensive and unified view of long-term trending availability and help maximize Pivotal Cloud Foundry deployment uptimes.

THE BENEFITS OF MONITORING PIVOTAL CLOUD FOUNDRY WITH VREALIZE OPERATIONS

Using vRealize Operations with vSphere adds critical monitoring that helps Pivotal Cloud Foundry maintain high availability:

- Detailed view of Pivotal Cloud Foundry key performance indicators
- Proactive identification and remediation of emerging performance, capacity, and configuration issues
- Comprehensive visibility across applications and infrastructure in a single console
- Automated capacity optimization and planning
- Unified view of availability for maximizing deployment uptime

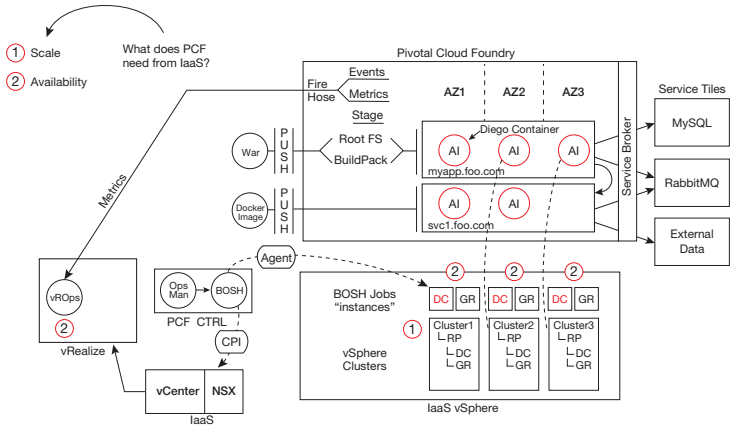

Figure 34: PCF and vSphere for availability.

Security

Lastly, Pivotal Cloud Foundry needs to run applications in a secure and auditable fashion, which can be a major challenge in a distributed environment where hundreds or thousands of applications can be running on a single PCF deployment.

The key here is to secure control of network access points and also be able to audit and report on all access as it occurs, both of which are enabled by vRealize in conjunction with the VMware NSX network virtualization and security platform.

PCF requires that an external load balancing service take in requests and forward them to the GoRouters, which then route requests to the application instances or AIs. The NSX Edge provides the load balancing services that PCF requires as well as offering network address translation (NAT) services and SSL cryptographic protocols for securing application traffic.

Providing Repeatability, Auditability, and Network Controls

NSX provides the key networking characteristics of a developer-ready infrastructure: repeatability, auditability, and software-defined network controls. NSX is repeatable in that all required network and security policies are driven by the API and automated. PCF and BOSH are capable of dynamically creating all required security principals within NSX to secure the platform. Network controls are also software defined and allow the platform operator to provide policy-driven, auditable controls but still allow developers self-service and agile consumption of PCF networking services.

Another aspect of the security services for a developer-ready infrastructure is provided by vRealize Log Insight. It ingests log information and events from applications, providing information not only about whether applications are being shut down, but also about who is shutting them down, and what is happening to the applications themselves—allowing an audit trail of the application's lifecycle and events.

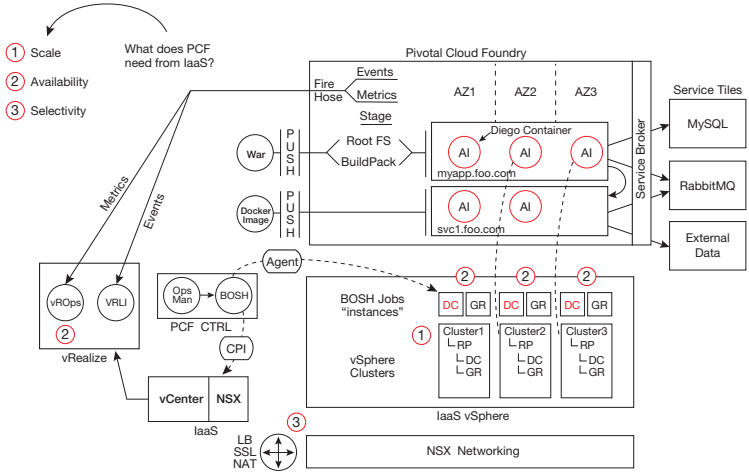

Figure 35: PCF and vSphere for security.

A Productive, Symbiotic Relationship

vSphere, NSX, vRealize Operations, and vRealize Log Insight work seamlessly with Pivotal Cloud Foundry to address issues of scalability, availability, and security. This ideal developer-ready infrastructure helps Pivotal Cloud Foundry run optimally in production.

The result puts infrastructure teams and application development teams in a productive, symbiotic relationship that returns immediate value for the business. IT teams get secure, software-driven compute, storage, and networking that is optimized for running, managing, and monitoring workloads at scale with minimal manual processes. Development teams receive common, shared tools that help them rapidly build cloud-native applications with agile processes and modern architectures.

Case Study: Optimizing Critical Banking Workloads

The financial services industry is constantly innovating to interact with their customers in new ways. At the same time, the IT environment must maintain a high level of economic efficiency, security, and robustness. The need for increased agility poses a challenge to the banking industry because it is subject to strict regulatory compliance.

Building on the proven foundation in server virtualization, VMware is enabling new consumption models, building on open technology like containers, while allowing customers to maintain a common platform and consistent operational model.

A leading financial group in Asia that serves more than four million customers with over 280 branches in 18 markets is, like so many other businesses, adapting to change. “The experience of telcos, transport, and retailing shows that we’re changing the way we communicate, the way we commute, and the way we consume. So why would banking be immune or be safeguarded from any of this?” the CEO of this financial group said.

Recognizing that change is required in order for this financial institution to maintain and expand their leadership in the industry, the company’s CIO gave the infrastructure team three goals:

1. Build a robust platform: Strengthen the technology and infrastructure platform to build world-class infrastructure
2. Be nimble: Develop solutions to support strategic priorities
3. Go explore: Nurture technology innovation to enhance customer experience

The team started by modernizing their data centers. Building on the existing vSphere foundation and following VMware Validated Design™ for vSphere metro-stretched clusters, they built new robust infrastructure to host their 4,000 legacy applications as well as applications being re-platformed from Solaris, AIX, and mainframe.

By design, for disaster avoidance, these clusters require resources available at all times in both primary and secondary data centers. This results in clusters being 30 to 40 percent utilized. The challenge was how to optimize cluster resource utilization across both of their data centers.

As part of their daily operations, this financial institution runs a lot of batch processing for applications such as grid computing and risk calculations. These applications typically required dedicated infrastructure that was maintained continuously but used sporadically, creating additional wasted resources.

As a result, the financial institution is striving to optimize resource utilization in existing infrastructure and reduce its infrastructure footprint for batch applications.

VMWARE FOOTPRINT

- Server virtualization technology
- vSphere Integrated Containers
- VMware Technical Account Manager Services

The Solution

SOLUTION

Re-platforming applications from legacy Unix to Linux, packaged in containers, and leveraging VMware vSphere® Integrated Containers™ to increase agility in scheduling batch jobs for business applications and improve resource utilization across clusters and data centers.

They started by looking for small applications that could be distributed and scaled horizontally. Guided by their VMware Technical Account Manager (TAM), they participated in the vSphere Integrated Containers Early Access Program. vSphere Integrated Containers has an opinionated provisioning model promoting strong isolation by provisioning containers as virtual machines (called container VMs). This financial institution found that, by containerizing the batch processing applications and running them with vSphere Integrated Containers, they were able to bring up capacity on-demand for any batch job, using the overhead capacity in their metro clusters. They used three criteria to identify candidates:

- Applications that do not readily support NAT'ing and require a unique routable IP address (with vSphere Integrated Containers, this can be achieved without the need for a network overlay because container images are instantiated as VMs, connected directly to vSphere Port Groups)
- Applications that need to be horizontally scaled up or down on demand (container VMs have access to vSphere cluster resources based on resource pool allocation allowing for dynamic resource allocation and resource balancing with vSphere Distributed Resource Scheduler™)
- Applications that require data persistence (vSphere Integrated Containers facilitates this by leveraging underlying vSphere storage)

This allowed this financial institution to change the VMware consumption model in ground-breaking manner. The consumption is now based on

dynamic quota assigned from the unused metro cluster capacity that can be provisioned directly via the Docker API. To quote the application team: "Get what you need when you need it. Then discard."

Business Results and Benefits

BUSINESS BENEFITS

- Accelerated application deployment from weeks to seconds.
- Reduced CapEx by eliminating legacy infrastructure necessary to run batch jobs.
- Reduced OpEx associated with operating systems maintenance and day-2 operations.

This financial institution is now running two business-critical applications through vSphere Integrated Containers and planning to expand to other similar batch type workloads. The ability to spin up the necessary applications on-demand in a predictable way allows the IT team to rapidly respond to the needs of the internal lines of business. In fact, they measured that deployment time for these batch applications servers went down from weeks to around 60 seconds. By leveraging the existing vSphere infrastructure and the well-recognized isolation boundary provided by the vSphere Virtual Machine, they were able to increase agility while maintaining the required level of governance and compliance.

On top of business agility, they were also able to bring savings to their infrastructure costs. They can now distribute the applications and have them share the same compute resources though the use of their batch scheduler. Where batch jobs used to have dedicated pools of resources, the load is now spread across the same pool of resources and scheduled on demand.

- Batch workloads are now running in vSphere alongside legacy workloads, and using all of the available capacity (50% overhead from metro clustering).
- vSphere Distributed Resource Scheduler (DRS) provides elastic resource management allowing them to schedule the excess metro cluster capacity in an efficient manner.
- Since vSphere Integrated Containers included in their vSphere licensing, it allows them to deploy this solution to all of their clusters without incurring additional licensing costs.

- Batch applications no longer incur overhead when not in use (before re-platforming, they would require dedicated infrastructure)

The solution also had a positive impact on their operational costs. The immutable and ephemeral quality of the Container VMs means that they were able to drastically reduce the operating systems maintenance and other day-2 operations costs. Because the container VMs are short-lived, it also allowed them to eliminate some of the security costs associated with agents licensing.

APPLICATIONS VIRTUALIZED

- Various business-critical batch applications required for risk calculations (hsVaR) And grid computing.
- Build Slaves for Jenkins software development process automation.

Looking Ahead

As the next step, this financial institution is planning to migrate the rest of the batch applications that are already targeted to move to this platform. More teams at the company are also experimenting with vSphere Integrated Containers. One application team is now using vSphere Integrated Containers to provision ephemeral Jenkins build slaves to optimize their continuous integration pipeline. Other applications being investigated include:

- Big data analytics: Spark compute with an object store as backend, spinning up Spark container VMs.
- Security scanning: Fortify port scanning requires resources on demand, which would fit well in the current model.

Conclusion

This book has introduced you to the world of cloud-native technology and practices. By explaining the business value of container technology, by exploring the use cases for containers and Kubernetes, and by demonstrating how to deploy cloud-native applications, this book primes you to begin adopting emerging technology to propel your organization into the digital era.

Indeed, the cloud-native elements covered in this book— such as containers, Kubernetes, microservices, container platforms, DevOps, and the CI/CD pipeline—converge into a powerful recipe for digital transformation: You can optimize the use of your computing resources and your software development practices to extend your enterprise's adaptability, productivity, innovation, competitive advantage, and global reach.

Glossary

This glossary presents definitions for terminology in the cloud-native space. The definitions are not intended to be axiomatic, dictionary-style definitions but rather plain-language descriptions of what a term means and an explanation of why the technology associated with it matters. For some of the terms, meaning varies by usage, situation, perspective, or context.

A

ACID: ACID stands for Atomicity, Consistency, Isolation, and Durability—properties of database transactions that, taken together, guarantee the validity of data in the face of power failures or system errors.

Active Directory: Microsoft Active Directory (AD) is a directory service that authenticates users and controls access to personal computers, servers, storage systems, applications, and other resources. An Active Directory domain controller combines a Kerberos key distribution center (KDC) with an LDAP server to provide authentication and authorization. To authenticate the identity of users, AD uses the highly secure Kerberos protocol or the legacy NT LAN Manager (NTLM). To authorize access to resources, AD typically uses a Privilege Attribute Certificate (PAC), which is a data structure in a Kerberos ticket that contains group memberships, security identifiers, and other information about a user's profile. See *LDAP*.

AKS: Azure Container Service (AKS) is Microsoft's managed Kubernetes service that runs in Azure.

API server: In Kubernetes, the API server provides a frontend that handles REST requests and processes data for API "objects," such as pods, services, and replication controllers.

Azure Container Registry. ACR is a private image registry from Microsoft that includes geo-replication.

B

build: With Docker, it is the process of building Docker images by using a Dockerfile. In the context of the CI/CD pipeline, the build process generates an artifact, such as a set of binary files that contain an application.

BOSH: An open source system that unifies release engineering, deployment, and lifecycle management for large distributed systems. BOSH performs monitoring, failure recovery, and software updates with zero-to-minimal downtime. Just as Kubernetes maintains the desire state of a containerized application, BOSH maintains the desired state of the underlying infrastructure, including Kubernetes itself, on which the application runs.

C

Cassandra: A NoSQL database, Apache Cassandra manages structured data distributed across commodity hardware. Common use cases include recommendation and personalization engines, product catalogs, play lists, fraud detection, and message analysis.

cloud computing: Cloud computing is an umbrella term for elastic, on-demand, shared computing resources and services—such as computational power, storage capacity, database usage, analytics, and software applications—delivered as a service over the Internet, typically with metered pricing. The organizations that provide cloud computing are frequently referred to as *cloud providers*. See also the definitions of the three cloud-computing service models: *IaaS*, *PaaS*, and *SaaS*. For a formal definition and taxonomy of cloud computing, see the *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, at https://csrc.nist.gov/publications/detail/sp/800-145/final.

Cloud Foundry Container Runtime. Formerly called Kubo, for Kubernetes on BOSH, CFCR is an open source project for deploying and managing Kubernetes by using BOSH. For more information on CFCR, see CloudFoundry.org. See also: *BOSH*.

cloud Infrastructure encompasses the servers, virtual machines, storage systems, networking, and other components required for cloud computing and infrastructure as a service. Cloud infrastructure provides the building blocks, or primitives, for creating hybrid and private clouds that deliver cloud computing services.

cloud-native applications: Generally speaking, cloud-native applications are apps that are developed and optimized to run in a cloud as distributed applications. More specifically, according to the Cloud Native Computing Foundation, cloud-native applications, which are also generally referred to as "modern" applications, are marked by the following characteristics:

- Containerized for reproducibility, transparency, and resource isolation.
- Orchestrated to optimize resource utilization.
- Segmented into microservices to ease modification, maintenance, and scalability.

Different organizations, however, have different definitions. Dell EMC, for example, defines *cloud-native application* as a highly scalable next-generation distributed application architecture that uses open standards and is dynamic in nature.

Cloud-native applications are typically developed and deployed on a containers as a service platform (CaaS) or a platform as a service (PaaS). Which see. See also: *12-factor app*.

Cloud Spanner: A globally distributed, strongly consistent database service that combines the benefits of a relational database structure with non-relational horizontal scale.

cluster: Three or more interconnected virtual machines or physical computers that, in effect, form a single system. A computer in a cluster is referred to as a node. An application running on a cluster is typically a distributed application because it runs on multiple nodes. By inherently providing high availability, fault tolerance, and scalability, clusters are a key part of cloud computing.

CNCF: Cloud Native Computing Foundation. An open source project hosted by the Linux Foundation, the CNCF hosts Kubernetes and other key open source projects, including Prometheus, OpenTracing, Fluentd, and linkerd. VMware is a member of the Linux Foundation and the Cloud Native Computing Foundation.

CNI: Container Network Interface. It is a open source project hosted by the CNCF to provide a specification and libraries for configuring network interfaces in Linux containers.

Concourse: Concourse is a system for continuous integration and continuous delivery that works with Pivotal Cloud Foundry and other platforms to help enterprise development teams release software early and often. Note that in the context of Concourse, the *D* in *CI/CD* stands for delivery, not deployment. Concourse automates the testing and packaging of frequent code commits. See *CI/CD*.

CoreDNS: An open source project, CoreDNS can integrate with Kubernetes, etcd, Prometheus, and other software to provide DNS and service discovery with plugins. CoreDNS is hosted by the CNCF.

container: A portable, executable format, known as an image, for packaging an application with all its dependencies and instructions on how to run it. When the container image is executed, it runs as a process on a computer or virtual machine with its own isolated, self-described application, file system, and networking. A container is more formally known as an *application container*. The use of containers is increasing because they provide a portable, flexible, and predictable way of packaging, distributing, modifying, testing, and running applications. Containers speed up software development and deployment.

containerize: To package an application in a container.

containerized application: An application that has been packaged to run in one or more containers.

containers as a service: A container-as-a service platform helps developers build, deploy, and manage containerized applications, typically by using Kubernetes or another orchestration framework, such as Mesos or Docker Swarm.

container host: A Linux operating system optimized for running containers. Examples include CoreOS and Project Photon OS by VMware.

container registry: See *registry*.

controllers: In Kubernetes, controllers are processes started by the Kubernetes Controller Manager to perform the routine tasks associated with managing a cluster.

CI/CD: Refers to either the continuous integration and continuous delivery pipeline or the continuous integration and continuous deployment pipeline. Context often, but not always, disambiguates the abbreviation.

See *continuous integration*, *continuous deliver*, and *continuous deployment*.

continuous integration constantly combines source code from different developers or teams into an app and then tests it.

continuous delivery readies an application or part of an application for production by packaging and validating it.

continuous deployment automatically deploys an application or part of an application into production.

converged infrastructure: Technology that brings together the disparate infrastructure elements powering IT, including servers, data storage devices, networking functions, virtualization, management software, orchestration, and applications. See *hyper-converged infrastructure*.

D

day one: Refers to deployment.

day two: Refers to post-deployment operations.

desired state: A key benefit of Kubernetes is that it automatically maintains the *desired state*—the state that an administrator or platform operator specifies an application should be in.

DevOps: Delivering software in an expedient, reliable, sustainable way requires collaboration between IT teams and developers. DevOps takes place when developers and IT come together to focus on operations in the name of streamlining and automating development and deployment. DevOps is a key practice driving the development and deployment of cloud-native applications.

developer-ready infrastructure: VMware vSphere, VMware NSX, VMware vSAN, and VMware vRealize Operations lays the foundation for a software-defined data center (SDDC). Running VMware Pivotal Container Service or Pivotal Cloud Foundry on top of a VMware SDDC, for example, produces developer-ready infrastructure—agile, self-service infrastructure that is ready to use to build and run cloud-native applications.

digital transformation: Optimizing the use of your computing resources, organizational processes, and software development practices to extend your enterprise's adaptability, productivity, innovation, competitive advantage, and global reach. At a high level, digital transformation often entails the adoption of new technologies, including cloud computing, mobile devices, social media, and big data analytics. At a lower level, cloud-native

technologies and practices—such as containers, Kubernetes, microservices, container platforms, DevOps, and the CI/CD pipeline—converge into a powerful recipe for digital transformation.

Docker is a widely used container format. Docker defines a standard format for packaging and porting software, much like ISO containers define a standard for shipping freight. As a runtime instance of a Docker image, a container consists of three parts:

- A Docker image
- An environment in which the image is executed
- A set of instructions for running the image

Docker Swarm is the name of a standalone native clustering tool for Docker. Docker Swarm combines several Docker hosts and exposes them as a single virtual Docker host. It serves the standard Docker API, so any tool that already works with Docker can transparently scale up to multiple hosts.

E

elastic: A resource or service that can dynamically expand or contract to meet fluctuations in demand.

ELK stack: Elasticsearch, Logstash, and Kibana combine to form the ELK stack. Taken together, these three open source projects provide a platform to collect, search, analyze, and visualize data. Elasticsearch is a distributed search and analytics engine that lets data engineers query unstructured, structured, and time-series data. Logstash lets you collect unstructured data, enrich it, and route it to another application, such as Elasticsearch. Kibana is a visualization engine to display data in dashboards as graphics and maps.

etcd: A distributed key-value store that Kubernetes uses to store data about its state and configuration.

F

fault tolerance: Fault tolerance is the property that lets a system continue to function properly in the event of component failure.

Fluentd: A data collector for unified logging. Fluentd, which works with cloud-native applications, is hosted by the CNCF.

G

GCP open service broker: It lets apps access Google cloud APIs from anywhere.

Gemfire: Pivotal Gemfire is a distributed data management platform that compresses operational data and holds it in memory to provide real-time, consistent, and scalable access to data-intensive NoSQL applications.

Google Cloud Platform: GCP.

Google Kubernetes Engine: It is a managed environment to deploy and scale containerized applications that are orchestrated by Kubernetes.

Greenplum Database: An ACID-compliant transactional database that employs a shared-nothing, massively parallel processing architecture, Pivotal Greenplum complies with SQL standards. It interoperates with industry-standard business intelligence and ETL tools as well as Hadoop. With a library of analytics functions and a framework for building custom functions, Greenplum addresses data warehousing use cases for big data.

GRPC: A project of the CNCF, GRPC is a open-source universal remote procedure call (RPC) framework for distributed systems. You can use it to define a service by using Protocol Buffers, a binary serialization language. GRPC also lets you automatically generate client and server stubs for a service in various languages.

H

Hadoop: Hadoop comprises the Hadoop Distributed File System (HDFS) and MapReduce. HDFS is a scalable storage system built for Hadoop and big data. MapReduce is a processing framework for data-intensive computational analysis of files stored in a Hadoop Distributed File System. Apache Hadoop is the free, open-source version of Hadoop that is managed by the Apache Software Foundation. The open-source version provides the foundation for several commercial distributions, including Hortonworks, IBM Open Platform, and Cloudera. There are also Hadoop platforms as a service. Microsoft offers HDInsight as part of its public cloud, Azure. Amazon Elastic MapReduce, or EMR, delivers Hadoop as a web service through AWS.

Harbor: An open source project from VMware formally known as Project Harbor, it is a secure registry that hosts repositories of container images.

Helm Chart: A package of Kubernetes resources that are pre-configured, customized, and reproducible; you can then manage a chart with the

Helm tool. The charts help improve the portability of Kubernetes applications. A single chart can contain an entire web application, including databases, caches, HTTP servers, and other resources.

horizontal pod autoscaler: In Kubernetes, a horizontal pod autoscaler is a controller that adds resources to handle an increase in demand when the requests to a service exceed the threshold set by the administrator.

hybrid cloud: Any modernized infrastructure that involves two or more delivery models, such as private cloud and public cloud resources.

hyper-converged infrastructure integrates the same key types of IT components that converged infrastructure does, but in a scalable rack or appliance that simplifies management, improves performance, and adds elastic scalability. See *converged infrastructure*.

I

image: With Docker, an image is the basis of a container. An image specifies changes to the root file system and the corresponding execution parameters that are to be used in the container runtime. An image typically contains a union of layered files systems stacked on top of each other. An image does not have state and it never changes.

infrastructure as a service (IaaS): Infrastructure-as-a-service (IaaS) provides on-demand access to underlying IT infrastructure, including resources for storage, networking, and compute. With IaaS, a user can provision IT services when they need them to deploy and run arbitrary software. Users typically pay only for the resources they consume. The user, however, does not manage or control the underlying cloud infrastructure. See *cloud computing*.

ingress: In Kubernetes, ingress refers to an API object that controls external access to the services in a Kubernetes clusters, such as HTTP and HTTPS. Ingress can perform load balancing.

J

Jaeger: A distributed tracing system released as open source software by Uber Technologies, Jaeger can monitor microservice-based architectures. Use cases include distributed transaction monitoring, root cause analysis, service dependency analysis, and performance optimization. Jaeger is hosted by the CNCF.

JSON: JavaScript Object Notation is a minimalist data-interchange format commonly used to annotate data, such as API output.

K

K8s: An abbreviation of sorts for Kubernetes.

KaaS: Kubernetes as a service.

Kafka: Apache Kafka partitions data streams and spreads them over a distributed cluster of machines to coordinate the ingestion of vast amounts of data for analysis. More formally, Kafka is a distributed publish-subscribe messaging system. A key use of Kafka is to help Spark or a similar application process streams of data. In such a use case, Kafka aggregates the data stream—for example, log files from different servers—into "topics" and presents them to Spark Streaming, which analyzes the data in real time.

kops: This term stands for Kubernetes Operations, a command-line tool to help you install, maintain, and upgrade Kubernetes clusters.

Kubernetes: An orchestration system that automates the deployment and management of containerized applications. As an application and its services run in containers on a distributed cluster of virtual or physical machines, Kubernetes orchestrates all the moving pieces to optimize the use of computing resources, to maintain the desired state, and to scale on demand. Kubernetes is also referred to as an orchestration framework or an orchestration engine. See *desired state* and *orchestration*.

kubectl: A command-line interface that you install on your computer and use to run commands that control and manage Kubernetes clusters.

kubelet: The agent that runs on each node in a Kubernetes cluster to manage pods. A PodSpec specifies how kubelet is to work. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the API server) and ensures that the containers described in those PodSpecs are running and healthy.

Kubo: See *Cloud Foundry Container Runtime*.

L

LDAP: Lightweight Directory Access Protocol. It is a standard protocol for storing and accessing directory service information, especially usernames and passwords. Applications can connect to an LDAP server to verify users and groups.

Lightwave: An open source security platform from VMware, Project Lightwave secures cloud platforms by providing a directory service,

Active Directory interoperability, Kerberos authentication, and certificate services. Lightwave empowers IT security managers to impose the proven security policies and best practices of on-premises computing systems on their cloud computing environment. More specifically, Lightwave includes the following services:

- Directory services and identity management with LDAP and Active Directory interoperability
- Authentication services with Kerberos, SRP, WS-Trust (SOAP), SAML WebSSO (browser-based SSO), OAuth/OpenID Connect (REST APIs), and other protocols
- Certificate services with a certificate authority and a certificate store

linkerd: A service mesh that adds service discovery, routing, failure handling, and visibility to cloud-native applications. linkerd is hosted by the CNCF.

M

Memcached: As a system that caches data in the distributed memory of a cluster of computers, Memcached accelerates the performance of web applications by holding the results of recent database calls in random-access memory (RAM).

microservices: A "modern" architectural pattern for building an application. A microservices architecture breaks up the functions of an application into a set of small, discrete, decentralized, goal-oriented processes, each of which can be independently developed, tested, deployed, replaced, and scaled. See *cloud-native application*.

micro-segmentation: With VMware NSX, micro-segmentation policies can specify granular traffic flow patterns among, for instance, the Kubernetes namespaces in which containerized applications are running. With micro-segmentation, you can craft rules that impose security requirements on workloads and isolate resources at the level of microservices.

Minikube: A tool that lets you run a single-node Kubernetes cluster inside a virtual machine or locally on a personal computer.

MongoDB: A distributed NoSQL document database, MongoDB stores data with a flexible, schema-free data model that can adapt to change. MongoDB includes secondary indexes, geospatial search, and text search. Common use cases include serving data to mobile applications and performing real-time analytics.

MySQL: It is an open source relational database management system (RDMS) that is commonly used in various types of applications, especially web apps. It is also widely embedded in the solutions distributed by independent software vendors (ISV) and original equipment manufacturers (OEM). In the name, SQL stands for Structured Query Language.

N

namespace: In the context of a Linux computer, a namespace is a feature of the kernel that isolates and virtualizes system resources. Processes that are restricted to a namespace can interact only with other resources and processes in the same namespace.

In Docker, namespaces isolate system resources like networking and storage.

In Kubernetes, when many virtual clusters are backed by the same underlying physical cluster, the virtual clusters are called namespaces.

NIST: The National Institute of Standards and Technology, which is part of the U.S. Department of Commerce. NIST publishes standards, guidelines, and requirements for information security.

NodePort: In Kubernetes, a NodePort presents a service, such as a web server, on a port on the nodes in a Kubernetes cluster for external access.

NoSQL: A NoSQL database stores data that is structured in a way other than the tabular relationships of traditional relational databases. NoSQL is also known as non-SQL, non-relational, and not-only SQL. NoSQL databases are commonly used for big data and real-time data processing. Popular examples of NoSQL databases include MongoDB, Cassandra, and Pivotal Gemfire.

NSX: VMware NSX is a product that provides software-defined network virtualization.

O

OCI stands for Open Container Initiative, an organization dedicated to setting industry-wide container standards. OCI was formed under the auspices of the Linux Foundation for the express purpose of creating open industry standards around container formats and runtime. The OCI contains two specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). VMware is a member of OCI. See https://www.opencontainers.org/.

OpenTracing: A vendor-neutral standard for distributed tracing. It is hosted by the CNCF.

opinionated platform: See *prescriptive platform*.

orchestration: Because it can automatically deploy, manage, and scale a containerized application, Kubernetes is often referred to as an orchestration framework or an orchestration engine. It orchestrates resource utilization, failure handling, availability, configuration, desired state, and scalability.

P

PaaS: See *platform as a service*.

PAS: Pivotal Application Service. Formerly known as Elastic Runtime, PAS runs Java, .NET, and Node apps on Pivotal Cloud Foundry.

PCF: Pivotal Cloud Foundry, a private platform as a service for developing and deploying cloud-native applications.

PKS: Pivotal Container Service, a Kubernetes-based container service.

Photon OS: An open source project from VMware, Project Photon OS is a Linux operating system optimized for running containers.

platforms: The overarching business objective of using a container platform is to accelerate the development and deployment of scalable, enterprise-grade software that is easy to modify, extend, operate, and maintain. Three types of platforms provide varying degrees of support for container technology:

- A platform for running individual container instances. A platform for running container instances helps developers build and test a containerized application. It does not, however, orchestrate the containerized application with Kubernetes, nor does it provide a service broker so that developers can integrate tools, databases, and services with an app. An example of a container instance platform is VMware vSphere Integrated Containers.
- Containers as a service.
- Platform as a service.

platform as a service: platform-as-a-service (PaaS) is a cloud-based environment for developing, testing, and running applications using programming languages, libraries, services, and tools supported or offered by the platform's provider. A platform as a service is sometimes referred

to simply as an application platform. In this context, an application platform helps developers not only write code but also integrate tools and services, such as a database, with their application as, for instance, microservices. An example of a private platform as a service that is also referred to as an application platform is Pivotal Cloud Foundry. See *containers as a service*, *infrastructure as a service*, and *cloud computing*.

platform developer: An engineer who customizes a Kubernetes platform (or another modern platform) to fit the needs of their project or organization.

platform operator: An engineer who manages a platform like Kubernetes.

pod: On Kubernetes, a pod is the smallest deployable unit in which one or more containers can be managed—in other words, you run a container image in a pod. A set of pods typically wraps a container, its storage resources, IP address, and other options up into an instance of an application that will run on Kubernetes. Docker is usually the container runtime used in a pod. A Kubernetes administrator or application developer specifies a pod by using a YAML file. Pods are commonly managed by a *deployment*, which see.

PostgreSQL: Also known as Postgres, it is an extensible object-relational database management system that securely stores data for large Internet-facing applications or data warehouses. Postgres is ACID-compliant; see *ACID*.

prescriptive platform: In the context of application platforms, a prescriptive platform hides the platform's complexity from developers by prescribing that developers use the system's formats, pipeline, and methods for building and running applications. For example, a prescriptive container platform might prescribe a scheduler, a runtime engine, integration with the underlying infrastructure, continuous delivery, and other aspects of the platform. A prescriptive platform is also referred to as an "opinionated" platform.

private cloud: A fully virtualized data center that includes two key capabilities that increase agility and are different from a virtualized data center: self-service and automation.

Prometheus: A open source monitoring system for Kubernetes. Prometheus is hosted by the CNCF.

pull: Downloading a container image from a registry into a local cache so that you can launch containers based on the image.

Q

quality of service: It is often abbreviated QoS.

R

RabbitMQ: An open source message broker, RabbitMQ implements the Advanced Method Queuing Protocol to give applications a common intermediate platform through which they can connect and exchange data.

RBAC: role-based access control. On Kubernetes, RBAC is a module that authorizes access to resources by role. RBAC empowers administrators to dynamically configure access policies through the Kubernetes API.

Redis: A key-value database, Redis can store a dataset in a networked, in-memory cache. Because keys in Redis can contain strings, hashes, lists, sets, sorted sets, bitmaps, and hyperlogs, Redis is often referred to as a data structure server. Data scientists, for instance, can perform operations on these data types to do things like compute set intersection, union and difference, and ranking.

registry: A hosted service that contains repositories of container images. Harbor, an open source project from VMware, is an example of a registry.

replica set: In Kubernetes, a replica set is a controller that manages the lifecycle of pods. See *controllers*.

repository: In the context of containers, a repository is a set of container images. The repository can be shared with other users through a registry server, and the images in the "repo" can be tagged with labels.

refactoring: Re-architecting an application or modifying its code to improve it. An application might, for example, be refactored by decomposing it into microservices.

repackaging: Placing a traditional application in a container format.

replatforming: Moving an application to another, more efficient platform. If the application being migrated is a traditional application and if the new platform uses containers, replatforming also involves repackaging.

rkt: Pronounced like *rocket*, rkt is a standards-based container engine from CoreOS.

runC: The code module that launches containers. It is part of containerd and managed by OCI, which stands for Open Container Initiative. See OCI.

S

scheduler: A module of a system or a software component that schedules and runs the deployment of containers, jobs, tasks, or another type of workload. Most public cloud services, such as Microsoft Azure, include a scheduler that lets you create jobs in the cloud. The jobs can, in turn, invoke services or tasks, such as backing up data or cleaning up logs.

service: The definition of *service* varies by context. In Kubernetes, it is an API object that describes how to access applications, such as a set of pods, by using methods like ports or load-balancers.

A service may also be a microservice within the context of some larger application. An HTTP server, for example, is a service.

service discovery: The automatic detection of services in a given context.

software-defined data center (SDDC): A data center in which infrastructure is virtualized and delivered as a service. The infrastructure of an SDDC includes virtualized networking and software-defined data storage and management. An SDDC supports applications in a way that is more flexible, agile, efficient, and cost-effective than traditional approaches. In a SDDC, all the components of infrastructure—compute, networking, storage, security, and availability—are abstracted and delivered as automated, policy-driven software. An SDDC radically reduces manual processes, speeds up IT service delivery, reduces costs, and improves ROI.

software as a service (SaaS): An application running on a cloud infrastructure that is used over a network, typically the Internet, instead of being downloaded and installed on local machines. The consumer of the service does not manage or control the underlying cloud infrastructure or the application's capabilities. Also known as a web app.

Spanner: See Cloud Spanner.

Spark: Apache Spark is an engine for large-scale data processing that can be used interactively from the Python shell. Spark combines streaming, SQL, and complex analytics by powering a stack of tools that can coexist in the same application. Spark can access diverse data sources, including not only the Hadoop File System (HDFS) but also Cassandra and MongoDB. Data scientists like Spark because they get access to Python's powerful numeric processing libraries.

spec: In Kubernetes, spec stands for specification. The specification is a description of a desired state, including the configuration supplied by a user.

Spring Cloud Data Flow: A toolkit for building data integration and real-time data processing pipelines. The Spring Cloud Data Flow server uses Spring Cloud Deployer to integrate pipelines with Pivotal Cloud Foundry, Mesos, or Kubernetes. Spring Cloud Data Flow helps engineers develop analytics pipelines by providing a distributed system that unifies ingestion, real-time analytics, batch processing, and data export.

StatefulSet: In Kubernetes, a StatefulSet manages the deployment and scaling of a set of pods according to your desired state. A stateful set can, for example, manage persistent storage and other resources for stateful pods.

swarm: With Docker, a swarm is a cluster of one or more Docker Engines running in swarm mode. Docker Swarm, however, is not the same thing as the swarm mode features in Docker Engine. See *Docker Swarm*.

T

tag: With Docker, a tag is a label that a user applies to a Docker image to distinguish it from other images in a repository.

the cloud: Computing resources available over the Internet. See *cloud computing*.

traditional application: A traditional application is monolithic in design with an n-tier application architecture that generally consists of database, application, and web servers. These components are usually tightly coupled with the infrastructure and dependent on it for high availability.

U

UID: It can stand for user identifier, user ID, or unique identifier, depending on the context or the system. With Kubernetes, for example, a UID is a string that uniquely identifies an object.

V

Vagrant: HashiCorp's Vagrant turns a machine's configuration into a distributable template to produce a predictable development environment for applications.

W

workload: A workload is the computational or transactional burden of a set of computing, networking, and storage tasks associated with an application. Similar apps with the same technology and tools can have radically different workloads under different circumstances or during

different times. Workloads can often be measured by CPU or memory consumption, network traffic, requests, database queries, transactions, and so forth. In very basic, broad terms, an application is a thing that processes something; a workload is the processing that's being done; and a use case is the reason that you do it. In the context of cloud computing and Kubernetes clusters, a workload can be seen as the amount of work that an instance of an app or part of an app performs during a certain time period.

X

XML: Extensible Markup Language. It is a flexible but verbose format for structuring and exchanging data. XML is often used in legacy applications, Java applications, and web applications for a variety of purposes, such as structuring configuration files or exchanging data. Although XML is sometimes used in cloud-native applications, JSON or YAML (which see) are the preferred data formats.

Y

YARN: A sub-project of Apache Hadoop, YARN separates resource management from computational processing to expand interactional patterns beyond MapReduce for data stored in HDFS. YARN allocates resources for Hadoop applications such as MapReduce and Storm as they perform computations. YARN, in effect, stands at the center of a Hadoop environment by providing a data operating system and pluggable architecture for other applications.

YAML: A human-readable data serialization standard commonly used in configuration files to structure information and commands. In Kubernetes, specification files are written in YAML.

volume: With Docker, a volume (or data volume) is a designated directory within one or more containers that bypasses the Union File System. Volumes are designed to persist data independent of the container's life cycle.

Z

ZooKeeper: Apache ZooKeeper coordinates distributed applications masquerading as animals. It provides a registry for their names. It configures and synchronizes them. It keeps them from running amok.

Numbers

12-factor app: A methodology for developing a software-as-a-service (SaaS) application—that is, a web app—and typically deploying it on a platform as a service or a containers as a service.

VMware, Inc.
3401 Hillview Avenue
Palo Alto CA 94304
USA Tel 877-486-9273
Fax 650-427-5001
www.vmware.com.