THE NEW STACK

vol. 4

NETWORKING, SECURITY & STORAGE WITH DOCKER & CONTAINERS

EDITED & CURATED BY ALEX WILLIAMS

The New Stack:
The Docker and Container Ecosystem Ebook Series

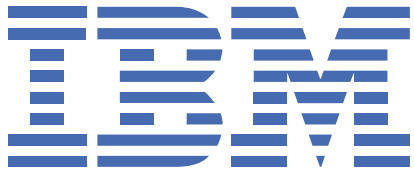
Alex Williams, Founder & Editor-in-Chief
Benjamin Ball, Technical Editor & Producer
Hoang Dinh, Creative Director
Lawrence Hecht, Data Research Director

Contributors:
Judy Williams, Copy Editor
Luke Lefler, Audio Engineer
Norris Deajon, Audio Engineer

TABLE OF CONTENT

SPONSORS

We are grateful for the support of the following ebook series sponsors:



And the following sponsors for this ebook:

INTRODUCTION

Keeping pace with the technology, practitioners and vendors in the container space is immensely difficult. It is the largest challenge in publishing our container ecosystem ebook series. Every time we narrow our area of focus, we've been opened up to yet another microcosm of experienced users, competing products and collaborative projects. Our solutions directory for the container ecosystem series has expanded with each book, and currently we have catalogued over 450 active products and projects. Calling this container technology space an ecosystem has become more and more accurate as adoption flourishes and the community makes greater strides.

Container technology has the ability to add so much speed to the development and deployment process, but deciding what option to choose from makes it difficult for potential adopters to get started. Comparatively, there are relative veterans who have long been composing applications with containers, have dealt with the software delivery pipeline, and automated much of the orchestration around containers. These practitioners are thinking more about how to securely network containers, maintain persistent storage, and scale to full production environments. With this ebook series, we look to educate both newcomers and familiars by going beyond operational knowledge and into analysis of the tools and practices driving the market.

Networking is a necessary part of distributed applications, and networking in the data center has only become more complex. In introducing container networking, we take a closer look at the demands that are driving this change in complexity, the evolution of types of container networks, the two primary container networking specifications, the role of software-defined networking, network configuration and service discovery, and networking with OpenStack.

It was also important for us to include a solid perspective on the best practices and strategies around container security. Container security has been an extremely compelling factor in this ebook series, and it's often been cited as a barrier to entry for containers. This ebook explains how containers can facilitate a more secure environment by addressing practices around security workflows. The components of this workflow include topics such as image provenance, security scanning, isolation and least privilege, auditing and more.

The book also explores how data storage fits into the dynamic and portable container lifecycle. We cover how to account for the temporary and portable nature of containers with strategies such as union filesystem architectures, host-based persistence, multi-host storage, volume plugins and software-defined storage. We discuss use cases for each of these storage strategies with the intent to show some of the patterns that have worked for others implementing container storage.

Networking, security and storage are all topics with broad and deep subject matter. Each of these topics deserves a full book of its own, but setting the stage in this initial ebook on these topics is an important exercise. The container ecosystem is becoming as relevant for operations teams as it is for developers who are packaging their apps in new ways. This combined interest has created a renaissance for technologists, who have become the central players in the emergence of new strategic thinking about how developers consume infrastructure.

There are more ways than one to skin a cat, and while we try to educate on the problems, strategies and products, much of this will be quickly outgrown. In two years' time, many of the approaches to networking, security and storage that we discuss in the ebook will not be as relevant. But the concepts behind these topics will remain part of the conversation. Containers will still need to communicate with each other securely,

container storage and security will need policy management, third-party storage and databases will need to be integrated so that stateful apps can run effectively, and so on.

While creating this ebook series we identified many additional topics worthy of their own book. So be on the lookout for more publications from us. In the meantime, please reach out to our team any time with feedback, thoughts, and ideas for the future. Thanks so much for your interest in our ebook series.

Thanks,

Benjamin Ball
Technical Editor and Producer
The New Stack

BRIDGING OPEN SOURCE AND CONTAINER COMMUNITIES

McGee talks about bringing together various tools in the open source and container ecosystems, including the many networking tools looking to address the needs of containers. IBM is focused on bringing these communities together by contributing to core technologies and building a world-class cloud platform. **Listen on SoundCloud** or **Listen on YouTube**

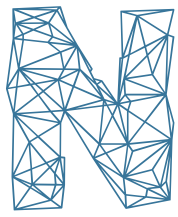
***Jason McGee**, IBM Fellow, is vice president and chief technical officer, Cloud Foundation Services. He is currently responsible for technical strategy and architecture across all of IBM Cloud, with specific focus on core foundational cloud services, including containers, microservices, continuous delivery and operational visibility services.*

Estes talks about the challenges of networking containers, the evolution of container namespaces, and the current state of container security, to which he offers a positive security outlook. This discussion extends into the plugin ecosystem for Docker, and how this pluggable model benefits the vendors and their customers.
Listen on SoundCloud or **Listen on YouTube**

***Phil Estes** is a senior technical staff in IBM's Cloud Open Technologies team, which leads IBM's strategy and involvement in key cloud open source technologies: Docker, Cloud Foundry and Openstack. Estes is a core contributor and maintainer on the Docker engine project.*

THE CONTAINER NETWORKING LANDSCAPE EXPLAINED

by **LEE CALCOTE**



etworking is an inherent component to any distributed application, and one of the most complicated and expansive technologies. As application developers are busily adopting container technologies, the time has come for network engineers to prepare for the unique challenges brought on by cloud-native applications.

With the popularization of containers and microservices, data center networking challenges have increased in complexity. The density by which containers are deployed on hosts (servers) presents challenges in terms of increased density of network interfaces. Operational complexity shifted from a few network interfaces on bare metal hosts, to a few network interfaces per virtual machine (VM) with twenty or so VMs per host, to a few interfaces per container with hundreds of containers per host.

Despite this increased density, the demands and measurements of reliability made of conventional networking hardware are the same demands and expectations made of container networking. Inevitably, operators will compare the performance of virtual machine networking to the performance of container networking and expect little difference — or

improved performance when containers are run directly on bare metal. Expectations of nominal care and feeding needed — or at least on par with their existing virtualized data center networking — are also mistakenly set.

This article is split into two primary areas of focus around types of container networking and container networking specifications. Networking starts with connectivity. Part one starts with the various ways in which container-to-container and container-to-host connectivity is provided.

This focuses on a breakdown of current container networking types, including:

- None
- Bridge
- Overlay
- Underlay

For the second half of this article, there are two container networking specifications that have set the stage for how the various components of container networking come together. These two specifications are:

- Container Network Model (CNM)
- Container Network Interface (CNI)

These specifications provide a mechanism for interoperability and extensibility as vendors adopt them. We'll explore these specifications as an understanding of their theory and flow of operation benefits operators greatly.

Types of Container Networking

While many gravitate toward network overlays as a popular approach to addressing container networking across hosts, the functions and types of container networking vary greatly and are worth better understanding as you consider the right type for your environment.

Some types are container engine-agnostic, and others are locked into a specific vendor or engine. Some focus on simplicity, while others on breadth of functionality or on being IPv6-friendly and multicast-capable. Which one is right for you depends on your application needs, performance requirements, workload placement (private or public cloud), etc. Let's review the more commonly available types of container networking.

Antiquated Types of Container Networking

The approach to networking has evolved as container technology advances. Two modes of networking have come and all but disappeared already.

Links and Ambassadors

Prior to having multi-host networking support and orchestration with Swarm, Docker began with single-host networking, facilitating network connectivity via `links` as a mechanism for allowing containers to discover each other via environment variables or `/etc/hosts` file entries, and transfer information between containers. The `links` capability was commonly combined with the ambassador pattern to facilitate linking containers across hosts and reduce the brittleness of hard-coded links. The biggest issue with this approach was that it was too static. Once a container was created and the environment variables defined, if the related containers or services moved to new IP addresses, then it was impossible to change the values of those variables.

Container-Mapped Networking

In this mode of networking, one container reuses (maps to) the networking namespace of another container. This mode of networking may only be invoked when running a **`docker`** container like this: **`--net:container:some_container_name_or_id.`**

This run command flag tells Docker to put this container's processes inside of the network stack that has already been created inside of another container. While sharing the same IP and MAC address and port numbers as the first container, the new container's processes are still confined to its own filesystem, process list and resource limits. Processes on the two containers will be able to connect to each other over the loopback interface.

This style of networking is useful for performing diagnostics on a running container and the container is missing the necessary diagnostic tools (e.g., curl or dig). A temporary container with the necessary diagnostics tools may be created and attached to the first container's network.

Container-mapped networking may be used to emulate pod-style networking, in which multiple containers share the same network namespace. Benefits, such as sharing localhost communication and sharing the same IP address, are inherent to the notion that containers run in the same pod, which is the behavior of **`rkt`** containers.

Current Types of Container Networking

Lines of delineation of networking revolve around IP-per-container versus IP-per-pod models and the requirement of network address translation (NAT) versus no translation needed.

None

None is straightforward in that the container receives a network stack, but lacks an external network interface. It does, however, receive a loopback

interface. Both the `rkt` and `docker` container projects provide similar behavior when none or null networking is used. This mode of container networking has a number of uses including testing containers, staging a container for a later network connection, and being assigned to containers with no need for external communication.

Bridge

A Linux bridge provides a host-internal network in which containers on the same host may communicate, but the IP addresses assigned to each container are not accessible from outside the host. Bridge networking leverages `iptables` for NAT and port-mapping, which provide single-host networking. Bridge networking is the default Docker network type (i.e., `docker0`), where one end of a virtual network interface pair is connected between the bridge and the container.

Here's an example of the creation flow:

1. A bridge is provisioned on the host.
2. A namespace for each container is provisioned inside that bridge.
3. Containers' ethX are mapped to private bridge interfaces.
4. iptables with NAT are used to map between each private container and the host's public interface.

NAT is used to provide communication beyond the host. While bridged networks solve port-conflict problems and provide network isolation to containers running on one host, there's a performance cost related to using NAT.

Host

In this approach, a newly created container shares its network namespace with the host, providing higher performance — near metal speed — and

eliminating the need for NAT; however, it does suffer port conflicts. While the container has access to all of the host's network interfaces, unless deployed in privilege mode, the container may not reconfigure the host's network stack.

Host networking is the default type used within Mesos. In other words, if the framework does not specify a network type, a new network namespace will not be associated with the container, but with the host network. Sometimes referred to as native networking, host networking is conceptually simple, making it easier to understand, troubleshoot and use.

Overlay

Overlays use networking tunnels to deliver communication across hosts. This allows containers to behave as if they are on the same machine by tunneling network subnets from one host to the next; in essence, spanning one network across multiple hosts. Many tunneling technologies exist, such as virtual extensible local area network (VXLAN).

VXLAN has been the tunneling technology of choice for Docker libnetwork, whose multi-host networking entered as a native capability in the 1.9 release. With the introduction of this capability, Docker chose to leverage HashiCorp's Serf as the gossip protocol, selected for its efficiency in neighbor table exchange and convergence times.

For those needing support for other tunneling technologies, Flannel may be the way to go. It supports **`udp`**, **`vxlan`**, **`host-gw`**, **`aws-vpc`** or **`gce`**. Each of the cloud provider tunnel types creates routes in the provider's routing tables, just for your account or virtual private cloud (VPC). The support for public clouds is particularly key for overlay drivers given that among others, overlays best address hybrid cloud use cases and provide scaling and redundancy without having to open public ports.

Multi-host networking requires additional parameters when launching the Docker daemon, as well as a key-value store. Some overlays rely on a distributed key-value store. If you're doing container orchestration, you'll already have a distributed key-value store lying around.

Overlays focus on the cross-host communication challenge. Containers on the same host that are connected to two different overlay networks are not able to communicate with each other via the local bridge — they are segmented from one another.

Underlays

Underlay network drivers expose host interfaces (i.e., the physical network interface at **`eth0`**) directly to containers or VMs running on the host. Two such underlay drivers are media access control virtual local area network (MACvlan) and internet protocol vlan (IPvlan). The operation of and the behavior of MACvlan and IPvlan drivers are very familiar to network engineers. Both network drivers are conceptually simpler than bridge networking, remove the need for port-mapping and are more efficient. Moreover, IPvlan has an L3 mode that resonates well with many network engineers. Given the restrictions — or lack of capabilities — in most public clouds, underlays are particularly useful when you have on-premises workloads, security concerns, traffic priorities or compliance to deal with, making them ideal for brownfield use. Instead of needing one bridge per VLAN, underlay networking allows for one VLAN per subinterface.

MACvlan

MACvlan allows creation of multiple virtual network interfaces behind the host's single physical interface. Each virtual interface has unique MAC and IP addresses assigned, with a restriction: the IP addresses need to be in the same broadcast domain as the physical interface. While many network engineers may be more familiar with the term subinterface (not to be confused with a secondary interface), the parlance used to describe

MACvlan virtual interfaces is typically upper or lower interface. MACvlan networking is a way of eliminating the need for the Linux bridge, NAT and port-mapping, allowing you to connect directly to the physical interface.

MACvlan uses a unique MAC address per container, and this may cause issue with network switches that have security policies in place to prevent MAC spoofing, by allowing only one MAC address per physical switch interface.

Container traffic is filtered from being able to speak to the underlying host, which completely isolates the host from the containers it runs. The host cannot reach the containers. The container is isolated from the host. This is useful for service providers or multi-tenant scenarios, and has more isolation than the bridge model.

Promiscuous mode is required for MACvlan; MACvlan has four modes of operation, with only the bridge mode supported in Docker 1.12. MACvlan bridge mode and IPvlan L2 mode are just about functionally equivalent. Both modes allow broadcast and multicast traffic ingress. These underlay protocols were designed with on-premises use cases in mind. Your public cloud mileage will vary as most do not support promiscuous mode on their VM interfaces.

A word of caution: MACvlan bridge mode assigning a unique MAC address per container can be a blessing in terms of tracing network traffic and end-to-end visibility; however, with a typical network interface card (NIC), e.g., Broadcom, having a ceiling of 512 unique MAC addresses, this upper-limit should be considered.

IPvlan

IPvlan is similar to MACvlan in that it creates new virtual network interfaces and assigns each a unique IP address. The difference is that the same MAC address is used for all pods and containers on a host — the

same MAC address of the physical interface. The need for this behavior is primarily driven by the fact that a commonly configured security posture of many switches is to shut down switch ports with traffic sourced from more than one MAC address.

Best run on kernels 4.2 or newer, IPvlan may operate in either L2 or L3 modes. Like MACvlan, IPvlan L2 mode requires that IP addresses assigned to subinterfaces be in the same subnet as the physical interface. IPvlan L3 mode, however, requires that container networks and IP addresses be on a different subnet than the parent physical interface.

802.1q configuration on Linux hosts, when created using `ip link`, is ephemeral, so most operators use network startup scripts to persist configuration. With container engines running underlay drivers and exposing APIs for programmatic configuration of VLANs, automation stands to improve. For example, when new VLANs are created on a top of rack switch, these VLANs may be pushed into Linux hosts via the exposed container engine API.

MACvlan and IPvlan

When choosing between these two underlay types, consider whether or not you need the network to be able to see the MAC address of the individual container. With respect to the address resolution protocol (ARP) and broadcast traffic, the L2 modes of both underlay drivers operate just as a server connected to a switch does, by flooding and learning using 802.1D packets. In IPvlan L3 mode, however, the networking stack is handled within the container. No multicast or broadcast traffic is allowed in. In this sense, IPvlan L3 mode operates as you would expect an L3 router to behave.

Note that upstream L3 routers need to be made aware of networks created using IPvlan. Network advertisement and redistribution into the

network still needs to be done. Today, Docker is experimenting with Border Gateway Protocol (BGP). While static routes can be created on top of the rack switch, projects like goBGP have sprouted up as a container ecosystem-friendly way to provide neighbor peering and route exchange functionality.

Although multiple modes of networking are supported on a given host, MACvlan and IPvlan can't be used on the same physical interface concurrently. In short, if you're used to running trunks down to hosts, L2 mode is for you. If scale is a primary concern, L3 has the potential for massive scale.

Direct Routing

For the same reasons that IPvlan L3 mode resonates with network engineers, they may chose to push past L2 challenges and focus on addressing network complexity in Layer 3 instead. This approach benefits from the leveraging of existing network infrastructure to manage the container networking. The container networking solutions focused at L3 use routing protocols to provide connectivity, which are arguably easier to interoperate with existing data center infrastructure, connecting containers, VMs and bare metal servers. Moreover, L3 networking scales and affords granular control, in terms of filtering and isolating network traffic.

Project Calico is one such project and uses BGP to distribute routes for every network — specifically to that workload using a /32 — which allows it to seamlessly integrate with existing data center infrastructure without the need for overlays. Without the overhead of overlays or encapsulation, the result is networking with exceptional performance and scale. Routable IP addresses for containers expose the IP address to the rest of the world; hence, ports are inherently exposed to the outside world.

Network engineers trained and accustomed to deploying, diagnosing and operating networks using routing protocols may find direct routing easier to digest. However, it's worth noting that Calico doesn't support overlapping IP addresses.

Fan Networking

Fan networking is a way of gaining access to many more IP addresses, expanding from one assigned IP address to 250 IP addresses. This is a performant way of getting more IPs without the need for overlay networks. This style of networking is particularly useful when running containers in a public cloud, where a single IP address is assigned to a host and spinning up additional networks is prohibitive, or running another load-balancer instance is costly.

Point-to-Point

Point-to-point is perhaps the simplest type of networking, and the default networking used by CoreOS rkt. Using NAT, or IP Masquerade (IPMASQ), by default, it creates a virtual ethernet pair, placing one on the host and the other in the container pod. Point-to-point networking leverages iptables to provide port-forwarding not only for inbound traffic to the pod, but also for internal communication between other containers in the pod over the loopback interface.

Capabilities

Outside of pure connectivity, support for other networking capabilities and network services needs to be considered. Many modes of container networking either leverage NAT and port-forwarding or intentionally avoid their use. IP address management (IPAM), multicast, broadcast, IPv6, load-balancing, service discovery, policy, quality of service, advanced filtering and performance are all additional considerations when selecting networking.

The question is whether these capabilities are supported and how developers and operators are empowered by them. Even if a container networking capability is supported by your runtime, orchestrator or plugin of choice, it may not be supported by your infrastructure. While some tier 2 public cloud providers offer support for IPv6, the lack of support for IPv6 in top public clouds reinforces the need for other networking types, such as overlays and fan networking.

In terms of IPAM, to promote ease of use, most container runtime engines default to host-local for assigning addresses to containers, as they are connected to networks. Host-local IPAM involves defining a fixed block of IP addresses to be selected. Dynamic Host Configuration Protocol (DHCP) is universally supported across container networking projects. Container Network Model (CNM) and Container Network Interface (CNI) both have IPAM built-in and plugin frameworks for integration with IPAM systems — a key capability to adoption in many existing environments.

Container Networking Specifications

There are two proposed standards for configuring network interfaces for Linux containers: The container network model (CNM) and the container network interface (CNI). As stated above, networking is complex and there are many ways to deliver functionality. Arguments can be made as to which one is easier to adopt than the next, or which one is less tethered to their benefactor's technology.

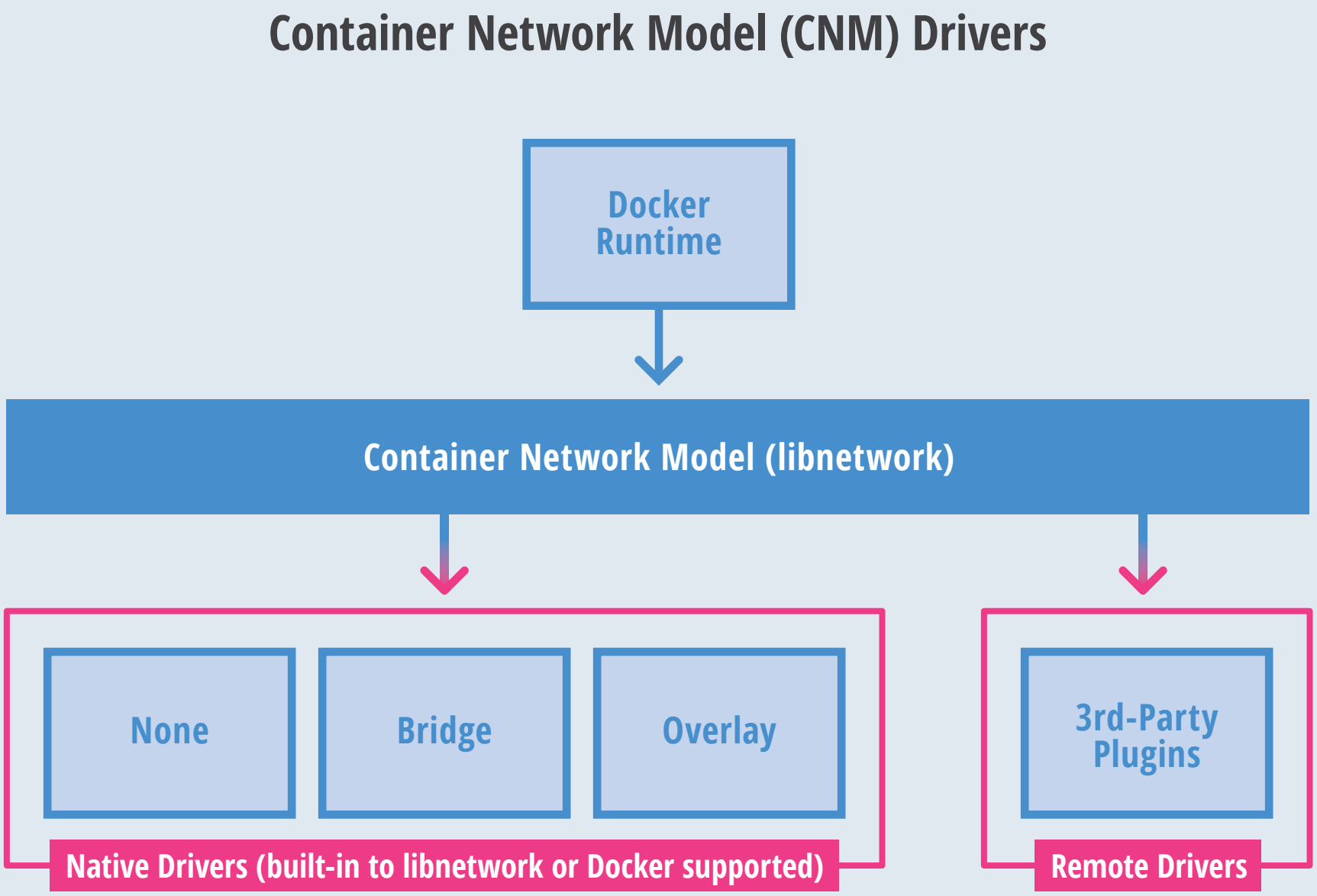
When evaluating any technology, some important considerations are community adoption and support. Some perspectives have been formed on which model has a lower barrier to entry. Finding the right metrics to determine the velocity of a project is tricky. Plugin vendors also need to consider the relative ease by which plugins may be written for either of these two models.

Container Network Model

The Container Network Model (CNM) is a specification proposed by Docker, adopted by projects such as libnetwork, with plugins built by projects and companies such as Cisco Contiv, Kuryr, Open Virtual Networking (OVN), Project Calico, VMware and Weave.

Libnetwork is the canonical implementation of the CNM specification. Libnetwork provides an interface between the Docker daemon and network drivers. The network controller is responsible for pairing a driver to a network. Each driver is responsible for managing the network it owns, including services provided to that network like IPAM. With one driver per network, multiple drivers can be used concurrently with containers connected to multiple networks. Drivers are defined as being either native (built-in to libnetwork or Docker supported) or remote (third-party

FIG 1: *Libnetwork provides an interface between the Docker daemon and network drivers.*



Source: Lee Calcote

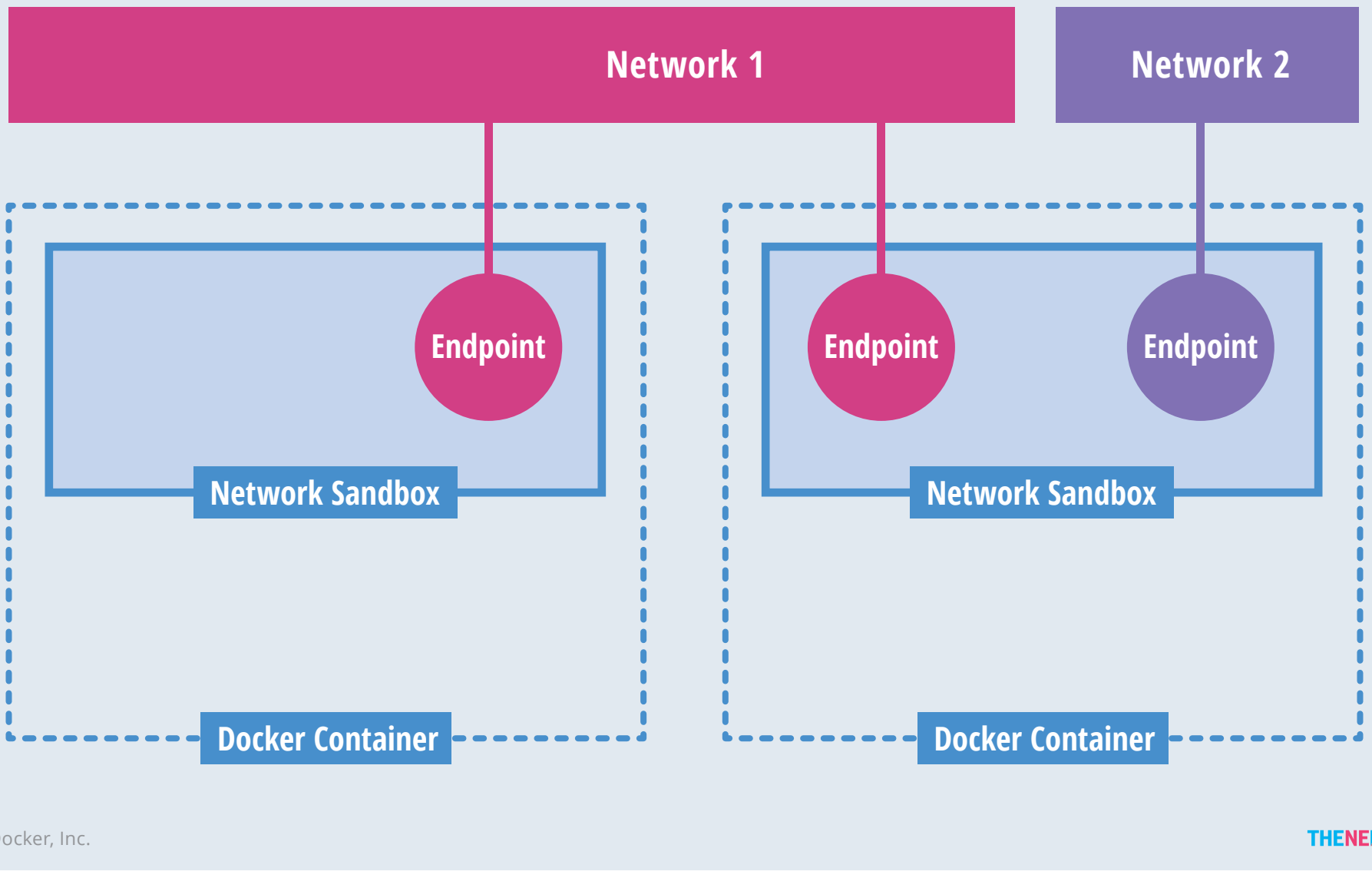


FIG 2: *Containers being connected through a series of network endpoints.*

plugins). The native drivers are none, bridge, overlay and MACvlan. Remote drivers may bring any number of capabilities. Drivers are also defined as having local scope (single host) or global scope (multi-host).

- **Network Sandbox:** Essentially the networking stack within a container, it is an isolated environment to contain a container's network configuration.
- **Endpoint:** A network interface that typically comes in pairs. One end of the pair sits in the network sandbox, while the other sits in a designated network. Endpoints join exactly one network, and multiple endpoints can exist within a single network sandbox.
- **Network:** A group of endpoints. A network is a uniquely identifiable group of endpoints that are able to communicate with each other.

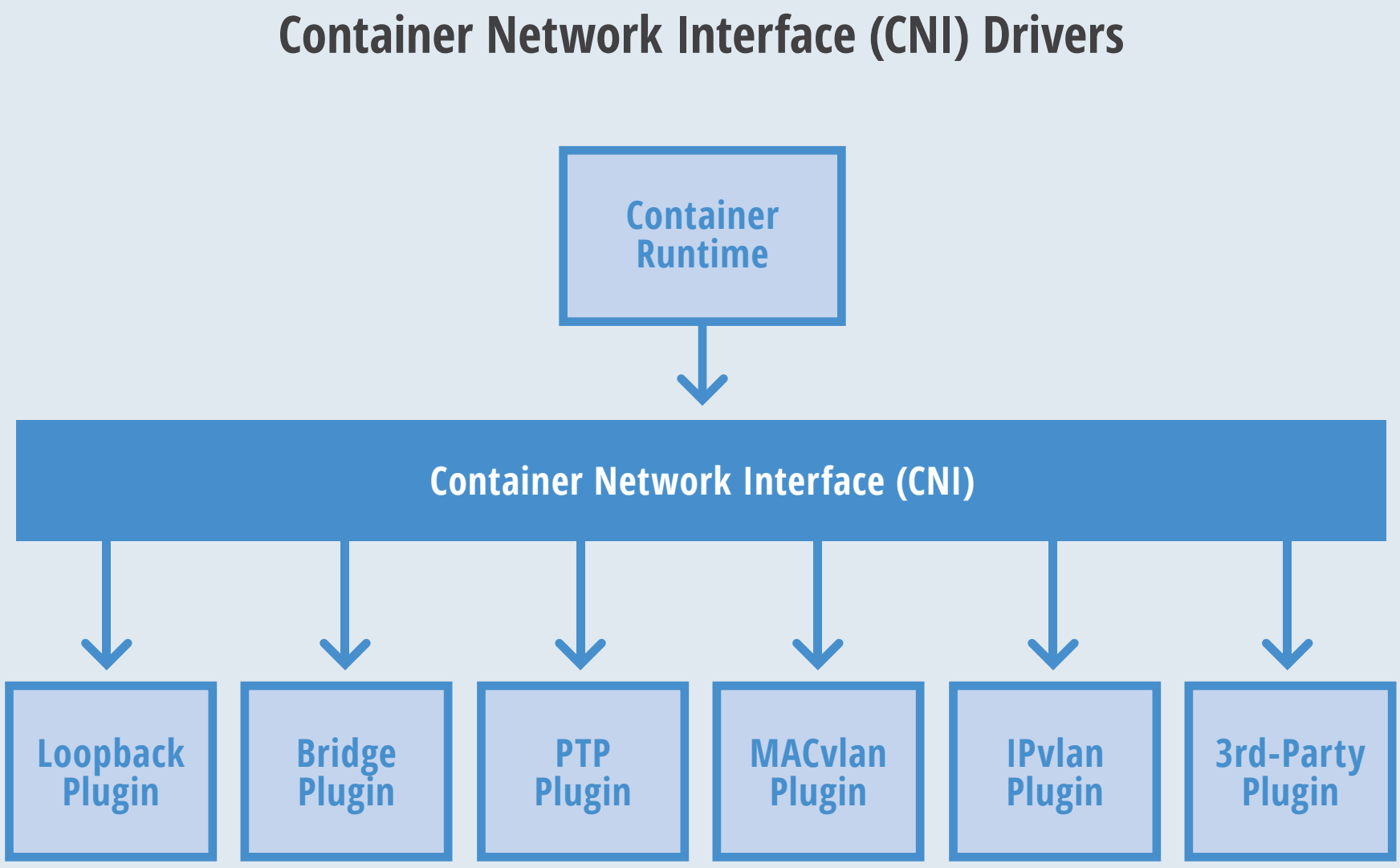
A final, flexible set of CNM constructs are options and labels (key-value pairs of metadata). CNM supports the notion of user-defined labels (defined using the `--label` flag), which are passed as metadata between libnetwork and drivers. Labels are powerful in that the runtime may inform driver behavior.

Container Network Interface

The Container Network Interface (CNI) is a container networking specification proposed by CoreOS and adopted by projects such as Apache Mesos, Cloud Foundry, Kubernetes, Kurma and rkt. There are also plugins created by projects such as Contiv Networking, Project Calico and Weave.

CNI was created as a minimal specification, built alongside a number of network vendor engineers to be a simple contract between the container runtime and network plugins. A JSON schema defines the expected input and output from CNI network plugins.

FIG 3: *CNI is a minimal specification for adding and removing containers to networks.*



Source: Lee Calcote

Multiple plugins may be run at one time with a container joining networks driven by different plugins. Networks are described in configuration files, in JSON format, and instantiated as new namespaces when CNI plugins are invoked. CNI plugins support two commands to add and remove container network interfaces to and from networks. `Add` gets invoked by the container runtime when it creates a container. `Delete` gets invoked by the container runtime when it tears down a container instance.

CNI Flow

The container runtime needs to first allocate a network namespace to the container and assign it a container ID, then pass along a number of parameters (CNI config) to the network driver. The network driver then attaches the container to a network and reports the assigned IP address back to the container runtime via JSON.

Mesos is the the latest project to add CNI support, and there is a Cloud Foundry implementation in progress. The current state of Mesos networking uses host networking, wherein the container shares the same IP address as the host. Mesos is looking to provide each container with its own network namespace, and consequently, its own IP address. The project is moving to an IP-per-container model and, in doing so, seeks to democratize networking such that operators have freedom to choose the style of networking that best suits their purpose.

Currently, CNI primitives handle concerns with IPAM, L2 and L3, and expect the container runtime to handle port-mapping (L4). From a Mesos perspective, this minimalist approach comes with a couple caveats, one of these being that the CNI specification does not specify any port-mapping rules to be used for a container; this capability may be handled by the container runtime. A second caveat is the fact that while operators should be allowed to change the CNI configuration, the behavior of container operation when CNI configuration is modified is not accounted for in the

specification. Mesos is addressing this ambiguity by ensuring that, upon restart of the CNI agent, they will checkpoint the CNI config when it is associated with the particular instance of the container.

CNM and CNI

In many respects these two container networking specifications democratize the selection of which type of container networking may be used, in that both are driver-based models, or plugin-based, for creating and managing network stacks for containers. Each allows multiple network drivers to be active and used concurrently, in that each provide a one-to-one mapping of network to that network's driver. Both models allow containers to join one or more networks. And each allows the container runtime to launch the network in its own namespace, segregating the application/business logic of connecting the container to the network to the network driver.

This modular driver approach is arguably more attractive to network operators than to application developers, in that operators are afforded the flexibility to select one or more drivers that deliver on their specific needs and fit into their existing mode of operation. Operators bear responsibility for ensuring service-level agreements (SLAs) are met and security policies are enforced.

Both models provide separate extension points, aka plugin interfaces, for network drivers — to create, configure and connect networks — and IPAM — to configure, discover, and manage IP addresses. One extension point per function encourages composability.

CNM does not provide network drivers access to the container's network namespace. The benefit here is that libnetwork acts as a broker for conflict resolution. An example conflict being when two independent network drivers provide the same static route, using the same route prefix,

but point to different next-hop IP addresses. CNI does provide drivers with access to the container network namespace. CNI is considering how it might approach arbitration in such conflict resolution scenarios in the future.

CNI supports integration with third-party IPAM and can be used with any container runtime. CNM is designed to support the Docker runtime engine only. With CNI's simplistic approach, it's been argued that it's comparatively easier to create a CNI plugin than a CNM plugin.

These models promote modularity, composability and choice by fostering an ecosystem of innovation by third-party vendors who deliver advanced networking capabilities. The orchestration of network micro-segmentation can become simple API calls to attach, detach and swap networks. Interface containers can belong to multiple networks, and each container can publish different services in different networks. The idea of different network constructs as first-class citizens is reflected in the ability to detach a network service from an old container and attach it to a new container.

Container Networking in OpenStack

Initially focused on infrastructure automation for virtual machines, OpenStack has come to focus on the needs of containers. Kuryr and Magnum are not their only container-related projects, but certainly the two concerned with container networking.

Kuryr

Kuryr, a project providing container networking, currently works as a remote driver for libnetwork to provide networking for Docker using Neutron as a backend network engine. Support for CNM has been delivered and the roadmap for this project includes support for CNI.

Magnum

Magnum, a project providing Containers as a Service (CaaS) and leveraging Heat to instantiate clusters running other container orchestration engines, currently uses non-Neutron networking options for containers.

Work is ongoing to integrate Kuryr and Magnum. The notion that containers may nest — recursively run — other containers creates some challenges for the project teams to overcome.

Networking with Intention

Intent-based networking defines the needs of network and network behavior in infrastructure-agnostic terms by using policy. With the increased density of network interfaces, volume of IP addresses, and complexity of communication across containers running interdependent microservices, networking stands to benefit from paradigms already defined and leveraged in other systems. Properties of intent-based networking draws from the successes of established concepts, referring to them by the terms invariant, portable, composable, scalable.

Invariant, for example, is similar to the idempotency concept in configuration management systems. Intent-based networking refers to this as invariant, meaning the that intent doesn't change as a result of intent policy being applied multiple times or by using different vendor equipment.

Another intent-based concept is portability, a concept characterized as policy supporting environments with heterogenous vendor equipment or policy that avoids any vendor-specific tethering. Intent-based networking is composable when policy can concurrently leverage disparate network services provided by different vendors' equipment or solutions.

The scalable quality of intent-based networking is self-explanatory; it enables each node or host to make networking decisions in a distributed fashion, with each node being aware of the intent-based policy. We've seen the power of constructs, like labels, tags and policies, significantly advance the capabilities of container networking technologies; intent-based networking stands to carry the ball further.

Summary

We discussed a number of considerations for choosing which type of networking to use in your environment — quite likely, you'll use a combination. Performance is certainly one of those considerations and will be the subject of further research. Outside of the various types of network connectivity and services, a significant consideration not highlighted is understanding to what extent you need to integrate with incumbent systems in your environment. For example, in the case of on-premises workloads, you'll have an existing IPAM solution. IPAM is provided by most container network vendors' drivers, but only some have integration with leading IPAM providers, such as Cisco, Infoblox, SolarWinds, etc.

As vendors and projects continue to evolve, the networking landscape continues to shift. Some offerings have consolidated or combined, such as Docker's acquisition of SocketPlane, and the transition of Flannel to Tigera — a new startup that has formed around Canal. Canal is a portmanteau of Calico and Flannel and a combination of those two projects. CoreOS will provide ongoing support for Flannel as an individual project, and will be integrating Canal with Tectonic, their enterprise solution for Kubernetes. Other changes come in the form of new project releases. Docker 1.12's release of networking features, including underlay and load-balancing support, is no small step forward for the project.

While there's a large number of container networking technologies and distinctly unique ways of approaching them, we're fortunate in that much of the container ecosystem seems to have converged and built support around only two networking models, at least for now. Developers would like to eliminate manual network provisioning in containerized environments, and barring those who have misconceptions of their job insecurity, network engineers are ready for the same.

Like other resources, an intermediary step to automated provisioning is pre-provisioning, meaning network engineers would preallocate networks with assigned characteristics and services, such as IP address space, IPAM, routing, QoS, etc., and developers or deployment engineers would identify and select from a pool of available networks in which to deploy their applications. Pre-provisioning needs to become a thing of the past, as we're all ready to move on to automated provisioning.

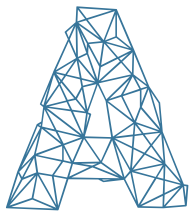
UNITING TEAMS WITH A DEVOPS PERSPECTIVE

In this discussion with Ken Owens of Cisco, we talk about how to apply existing models of networking, security and storage to containerized environments. Owens talks about how bringing a DevOps mindset forward to containers reveals the strong foundations for security, networking and storage and how they still apply. But the challenge is often in the varying perspectives involved in operating these environments, from Linux and network administrators to security and storage teams. It's important to link up these perspectives through the components they control, while creating policy around resource management. The discussion moves on to the roles of Contiv and Mantl, and how they address these issues. **Listen on SoundCloud** or **Listen on YouTube**

***Ken Owens** is chief technical officer of Cloud Infrastructure Services (CIS) at Cisco Systems. Owens is responsible for creating and communicating both technical and scientific vision and strategy for CIS business. He brings a compelling view of the technology trends in enterprise IT (e.g., infrastructure, computing, SaaS, virtualization and cloud) and evangelizes the technology roadmap for the business. Before joining Cisco in 2014, Ken spent over seven years at Savvis as the chief scientist, CTO and vice president of Security and Virtualization Technologies.*

THREE PERSPECTIVES ON NETWORK EXTENSIBILITY

by **SCOTT FULTON III**



critical aspect of any cloud-based deployment is managing the networking between the various components of the workload. In this chapter, we'll present perspectives on three styles of integrated container networking by way of plugins. The previous section introduced the Container Network Model (CNM) and the Container Network Interface (CNI). We'll discuss the origin of these models, as well as a third area that includes the Apache Mesos ecosystem.

Container Network Model and Libnetwork

Docker's extensibility model adds capability to the daemon in the way a library adds capability to an operating system. It involves a code library, similar to Docker's runtime, but used as a supplement. That library is called libnetwork, produced as a third-party project by the development team SocketPlane, which Docker Inc. acquired in March 2015.

Essentially, libnetwork provides a kind of plank on which developers may write network drivers. The binding principle of libnetwork is called the

Container Network Model (CNM), which was conceived as a container's bill of rights. One of those rights is equal access to all other containers in a network; partitioning, isolation, and traffic segmentation are achieved by dividing network addresses. A service discovery model provides a means for containers to contact one another.

The intention is for libnetwork to implement and use any kind of networking technology to connect and discover containers. It does not specify one preferred methodology for any network overlay scheme. Project Calico is an example of an independent, open source project to develop a vendor-neutral network scheme for Layer 3; developers have recently made Project Calico's calicoctl library an addressable component of a Docker plugin.

ClusterHQ was one of the first companies to implement a persistent container system for databases, called Flocker. It uses libnetwork, and is addressable using Weaveworks' Weave Net overlay. As ClusterHQ Vice President of Product Mohit Bhatnagar told us:

"I think we are at a point where customers who initially thought of containers for stateless services need to realize both the need and the potential for stateful services. And we are actually very pleasantly surprised about the number of customer engagements ... regarding stateful services."

The critical architectural distinction for the Docker scheme concerns just what part is being extended. In Docker architecture, the daemon of Docker Engine runs on the host server where the applications are being staged. Docker Swarm reconfigures Docker Engine's view of the network, replacing it with an amalgamated view of servers running in a cluster. Swarm is effectively the orchestrator, but plugins can extend Docker Engine at a lower layer.

Container Network Interface

Kubernetes published guidelines for implementing networked extensibility. It should be capable of addressing other containers' IP addresses without resorting to network address translation (NAT), and should permit itself to be addressed the same way. Essentially, as long as the component is addressable with IP, Kubernetes is fine with it. In that context, theoretically anything could extend what you do with Kubernetes, but nothing had to be bound to it.

"We looked at how we were going to do networking in Kubernetes," explained Google Engineering Manager Tim Hockin, "and it was pretty clear that there's no way that the core system could handle every network use case out there. Every network in the world is a special snowflake; they're all different, and there's no way that we can build that into our system. We had to externalize it, and plugins are the way we're doing that."

Then CoreOS produced its Container Network Interface (CNI). It's more rudimentary than CNM, in that it only has two commands: create a container and remove a container. Configuration files written in JSON instantiate the container's contents and set it up with an IP address. But since that address follows Kubernetes' guidelines, Google decided it's fine with CNI. As a result, Flannel and Weave Net have been implemented as Kubernetes plugins using CNI.

Hockin acknowledged that while such extensions enable new and flexible forms of networking into a Kubernetes environment, they also incur some costs. "The general Kubernetes position on overlays is, you should only use them if you really, really have to. They bring their own levels of complexity and administration and bridging. We're finding that more and more users of Kubernetes are going directly to L3 routing, and they're having a better time with that than they are with overlays."

After re-evaluating the current state of extensibility frameworks, ClusterHQ concluded that which model you choose will depend, perhaps entirely, on how much integration you require between containers and pre-existing workloads.

"If your job previously ran on a VM, and your VM had an IP and could talk to the other VMs in your project," explained ClusterHQ Senior Vice President of Engineering and Operations Sandeepan Banerjee, "you are probably better off with the CNI model and Kubernetes and Weave." Banerjee then cited Kubernetes' no-NAT stipulation as the key reason.

"If that is not a world that you are coming from," he continued, "and you want to embrace the Docker framework as something you see as necessary and sufficient for you across the stack — including Docker's networking library, Swarm as an orchestration framework, and so on — then the Docker proposal is powerful, with merits, with probably a lot more tunability overall."

Mesosphere and Plugins from the Opposite End

Mesosphere has produced perhaps the most sophisticated commercial implementation of Mesos with DC/OS, and has built on top an effective competitor against Kubernetes in the form of its Marathon orchestrator.

As a scheduling platform, the job of extending the reach of Mesos has historically been done from the opposite side of the proverbial bridge. Enabling scheduling for big data processes in Hadoop, job management processes in Jenkins, and container deployment in Docker, has all been done from within those respective platforms.

But in the summer of 2016, Mesosphere began to take a different stance. Mesosphere began to pave the way for properly interfaced containers to

extend this library by way of CNI. At the time of this writing, Mesosphere had published a document stating its intent to implement CNI support in the near term of the DC/OS roadmap.

"We're in a world today where there's enough different vendors out there, with varying interfaces and implementations for networking and storage," said Ben Hindman, founder and chief architect at Mesosphere, "that the means of doing plugins, I think, is a pretty important part. What I think is not so clear now, is whether or not the plugins that were defined by Docker will become the universal plugins. And I think what you're seeing in the industry already today is, that's not the case."

Currently, DC/OS uses an open source load balancing and service discovery system called Minuteman to connect containers to one another. It works by intercepting packets as they're being exchanged from a container on one host to one on a different host, and rewriting them for the proper destination IPs. This accomplishes the cross-cloud scope that distinguishes DC/OS from other implementations. Alternately, DC/OS offers a mechanism for setting up a virtual extensible LAN (VXLAN), and establishing routing rules between containers in that virtual network. Mesosphere does not reinvent the wheel here at all; actually, it gives users their own choice of overlay schemes, based on performance or other factors.

Hindman told us he sees value in how Flannel, Weave, and other network overlay systems solve the problem of container networking, at a much higher level than piecing together a VXLAN. The fact that such an alternative would emerge, he said, "is just capturing the fact that we, as an industry, are sort of going through and experimenting with a couple of different ways of how we might want to do stuff. I think that we'll probably settle on a handful of things, and overlays are still going to be there. But there are going to be some other ways in which people link together and

connect up containers that are not using pre-existing, SDN-based technologies."

Integration Towards the Future

Today, containerization is not often found as a line item in IT and data center budgets — integration is. When the people signing the checks don't quite understand the concepts behind the processes they are funding, integration often provides them with as much explanation as they require to invest both their faith and their capital expense. Some might think this is a watering down of the topic. In truth, integration is an elevation of the basic idea to a shared plane of discussion. Everyone understands the basic need to make old systems coexist, interface and communicate with new ones. So even though the methodologies may seem convoluted or impractical in a few years, the inspiration behind working toward a laudable goal will have made it all worth pursuing.

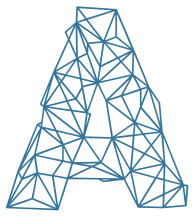
AN AUTOMATED MODEL FOR CONTAINER SECURITY

In this discussion with John Morello of Twistlock, we talk about how containers can actually be a better medium for automating and securing applications. Containers being immutable and lightweight makes it easier to follow images from early in the development life cycle all the way to the registry and compute environments. Twistlock collects data from this life cycle and creates a predictive model for a container's behavior. This model looks for inconsistent behaviors, and depending on what you want, it can set off an alert or even block the activity entirely. Later in the discussion, we talk about Twistlock's focus on four distinct use cases, recent changes to its core features, the value of partner integration and more. **Listen on SoundCloud** or **Listen on YouTube**

John Morello *is the chief technology officer (CTO) of Twistlock. As CTO, Morello leads the work with strategic customers and partners, and drives the product roadmap. Prior to Twistlock, John was the chief information security officer (CISO) of Albemarle, a Fortune 500 global chemical company where Morello was responsible for servers, networking, security and devices globally. Prior to that, Morello spent 15 years in Microsoft Consulting Services, where he built solutions for enterprises and governments around the world.*

ASSESSING THE CURRENT STATE OF CONTAINER SECURITY

by **ADRIAN MOUAT**



ny rational organization that wishes to run mission-critical services on containers will at some point ask the question: “But is it secure? Can we really trust containers with our data and applications?”

Amongst tech folks, this often leads to a containers versus virtual machines (VMs) debate and a discussion of the protection provided by the hypervisor layer in VMs. While this can be an interesting and informative discussion, containers versus VMs is a false dichotomy; concerned parties should simply run their containers inside VMs, as currently happens on most cloud providers. A notable exception is Triton from Joyent, which uses SmartOS Zones to ensure isolation of tenants.

There is also a growing community who believe that container security and isolation on Linux has improved to the point that one can use bare metal container services not using VMs for isolation; for example, IBM has built a managed container service on the public Bluemix cloud service that is running without VM isolation between tenants.

To retain the agility advantage of containers, multiple containers are run within each VM. Security conscious organizations may use VMs to separate containers running at different security levels; for example, containers processing billing information may be scheduled on separate nodes to those reserved for user facing websites. Several companies — including Hyper_, Intel and VMware — are working on building lightning-fast VM-based frameworks that implement the Docker API in an attempt to marry the speed of container workflows and hypervisor security.

Once we accept that moving to containers does not imply surrendering the established and verified security provided by hypervisors, the next step is to investigate the security gains that can be achieved through the use of containers and a container-based workflow.

Security Workflow

In a typical workflow, once the developer has completed a feature, they will push to the continuous integration (CI) system, which will build and test the images. The image will then be pushed to the registry. It is now ready for deployment to production, which will typically involve an orchestration system such as Docker's built-in orchestration, Kubernetes, Mesos, etc. Some organizations may instead push to a staging environment before production.

In a system following security best practices, the following features and properties will be present:

- **Image Provenance:** A secure labelling system is in place that identifies exactly and incontrovertibly where containers running in the production environment came from.
- **Security Scanning:** An image scanner automatically checks all images for known vulnerabilities.

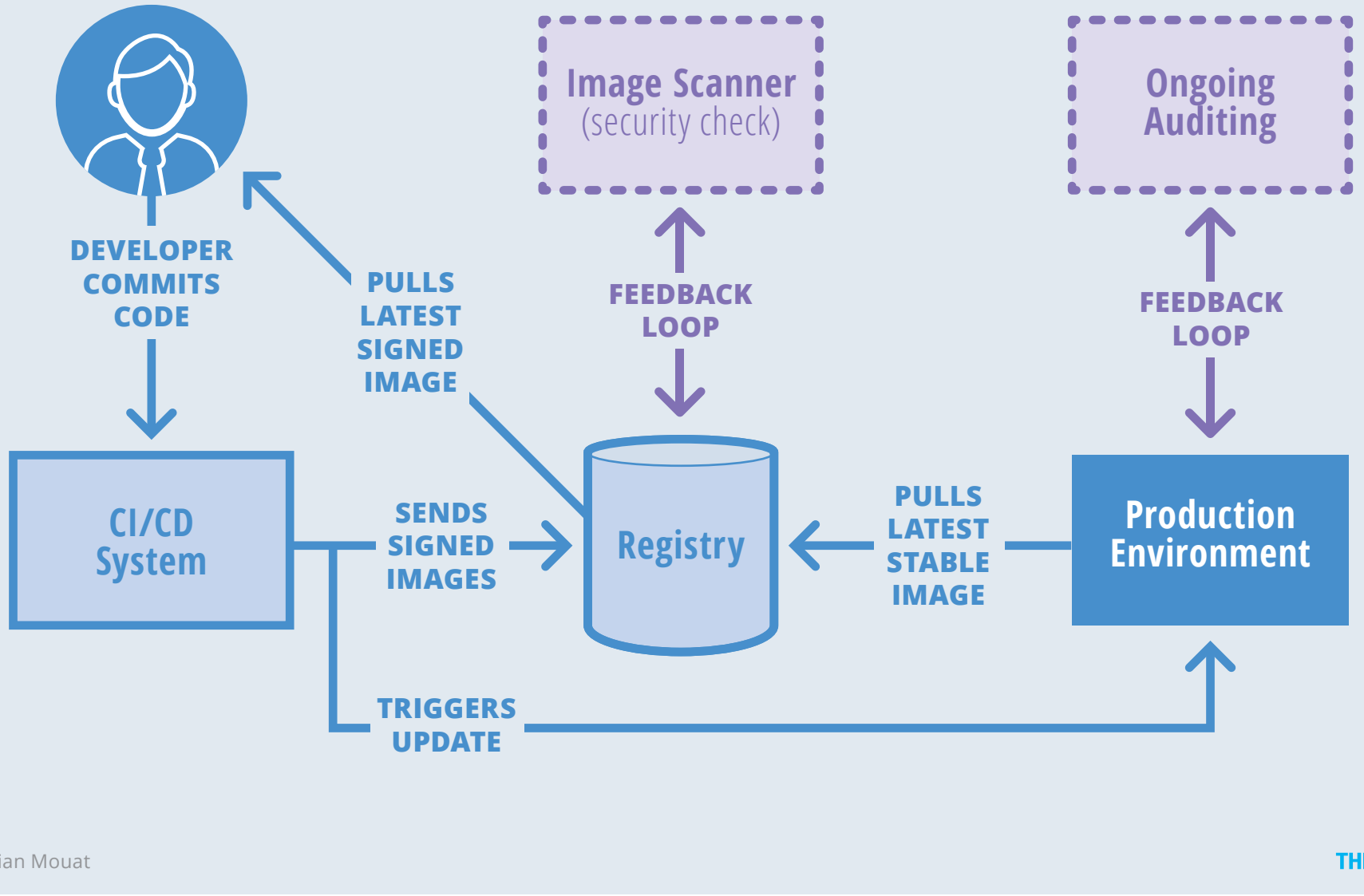


FIG 1: *In a container workflow, developers will push changes to a CI system with a testing component. Signed images will move to the registry, where it can be deployed to production.*

- **Auditing:** The production environment is regularly audited to ensure all containers are based on up-to-date containers and both hosts and containers are securely configured.
- **Isolation and Least Privilege:** Containers run with the minimum resources and privileges needed to function effectively. They are not able to unduly interfere with the host or other containers.
- **Runtime Threat Detection and Response:** A capability that detects active threats against a containerized application in runtime and automatically responds to it.
- **Access Controls:** Linux security modules, such as AppArmor or SELinux, are used to enforce access controls.

Image Provenance

Organizations need to be careful about the software they are running, especially in production environments. It is essential to avoid running out-of-date, vulnerable software, or software that has been compromised or tampered with in some way. For this reason, it is important to be able to identify and verify the source of any container, including who built it and exactly which version of the code it is running.

The gold standard for image provenance is Docker Content Trust. With Docker Content Trust enabled, a digital signature is added to images before they are pushed to the registry. When the image is pulled, Docker Content Trust will verify the signature, thereby ensuring the image comes from the correct organization and the contents of the image exactly match the image that was pushed. This ensures attackers did not tamper with the image, either in transit or when it was stored at the registry. Other, more advanced, attacks — such as rollback attacks and freeze attacks — are also prevented by Docker Content Trust, through its implementation of The Update Framework (TUF).

At the time of writing, Docker Content Trust is supported by Docker Hub, Artifactory and the Docker Trusted Registry (currently experimental). It is possible to setup the open source private registry with Docker Content Trust, but this requires also standing up a Notary server (see these instructions for more details).

In the absence of Docker Content Trust, it is still possible to verify image provenance using digests, which are cryptographic hashes of the contents of an image. When an image is pushed, the Docker client will return a string (such as sha256:45b23dee09af...6677c5cb2) that represents the digest of the image. This digest can then be used to pull the image. Whenever an image is pulled in this manner, Docker will verify the digest

matches the image. Unlike tags, a digest will always point to exactly the same image; any update to the image will result in the generation of a new digest. The problem with using digests is organizations need to set up a proprietary system for automatically extracting and distributing them.

The provenance of images from third-parties — whether they are used directly or as base images — also needs to be established. When using Docker Hub, all "official" images have been vetted by Docker, have content trust information attached and should be considered the safest Hub images. Discretion should be applied when using other images, but note that "automated builds" are linked to the source code they are built from, and should be considered more trustworthy than regular user images. Organizations should consider building images from source themselves rather than pulling from untrusted repositories. This situation is currently changing somewhat with the emergence of Docker Store, which will provide a trusted store for publishers along the same lines as Apple's App Store.

Security Scanning

Security scanning of Docker images is a new service being offered by several companies. The basic idea is simple: take a Docker image and cross-reference the software it contains against a list of known vulnerabilities to produce a "bill of health" for the image. Based on this information, organizations can then take action to mitigate vulnerabilities.

The current offerings include Atomic Scan from Red Hat, Bluemix Vulnerability Advisor from IBM, Clair from CoreOS, Docker Security Scanning from Docker Inc., Peekr from Aqua Security, and Twistlock Trust. They vary widely in how they work, how they are accessed and how much they cost. One crucial difference is the way in which the scanners identify software installed in images.

Some scanners, including Clair, will just interrogate the package manager (e.g., Apt on Debian and Ubuntu) to find the installed software, but this won't work for software installed through tarballs, or with package managers the scanner doesn't recognize. In contrast, Docker Security Scanning performs a binary-level analysis of images that works regardless of the package manager and can also identify versions of statically-linked libraries. Twistlock is also interesting in that it performs scanning on software installed through tarball, features zero-day feeds in their vulnerability scanning, and works in air-gapped environments.

It is essential to consider how security scanning can be integrated into your systems. Docker Security Scanning is available as an integrated part of Docker Cloud and Docker Datacenter, but not as a stand-alone service. Other providers offer an application program interface (API), allowing integration into existing CI systems and bespoke workflows. Some scanners can be installed on-premises, which will be important to organizations with a requirement to keep all software within their boundaries.

Once you've integrated a security scanning service, your first thought may be to have a blanket ban on running any images with vulnerabilities in production. Unfortunately, you are likely to find that most of your images have some vulnerabilities and this isn't a realistic option. For example, Ubuntu has one of the best records for quickly updating images, but — at the time of writing — the 16.04 base image ships with a single major vulnerability due to the version of Perl used (most other images have considerably more issues). Therefore, you are likely to find that you need to investigate discovered vulnerabilities individually to ascertain whether or not they represent a real risk to your system.

This situation can be alleviated significantly by using lightweight containers that have unnecessary software stripped out. The simplest way

to do this is to use a very small base image, such as Alpine, which comes in at only 5MB. Another, somewhat extreme, possibility is to build a statically linked binary and copy it on top of the empty "scratch" image. That way there are no OS level vulnerabilities at all. The major disadvantage of this approach is that building and debugging become significantly more complex — there won't even be a shell available.

Automated scanning is a huge move forward for security in our industry. It quickly surfaces potential risks and places pressure on vendors to patch vulnerable base images in a timely manner. By paying attention to the results of scans and reacting quickly, organizations can stay one step ahead of many would-be attackers.

Auditing

Auditing directly follows security scanning and image provenance. At any point in time, we would like to be able to see which images are running in production and which version of the code they are running. In particular, it is important to identify containers running out-of-date, potentially vulnerable images.

When working with containers, it is strongly recommended to follow what is sometimes called a "golden image" approach: do not patch running containers, but instead replace them with a new container running the updated code — blue-green deployments and rolling upgrades can be used to avoid downtime. With this approach, it is possible to audit large numbers of running containers by looking at the image they were built from. Tools, such as docker diff, can be used to verify that container filesystems have not diverged from the underlying image.

Note that it isn't enough to scan images before they are deployed. As new vulnerabilities are reported, images with a previous clean bill of health will

become known-vulnerable. Therefore, it is important to keep scanning all images that are running in production. Depending on the scanning solution used, this doesn't necessarily involve an in-depth rescan; scanners can store the list of software from scanned images and quickly reference this against new vulnerabilities.

It is still important to audit the hosts in a container-based system, but this can be made easier by running a minimal distribution, such as CoreOS, Red Hat Atomic or Ubuntu Snappy, which are designed to run containers and simply contain less software to audit. In addition, tools, such as Docker Bench for Security, can be used to check configurations, and both Aqua Security and Twistlock offer solutions that audit hosts and configurations.

Isolation and Least Privilege

A major security benefit of containers is the extra tooling around isolation. Containers work by creating a system with a separate view of the world — separate namespaces — with regards to the filesystem, networking and processes. In addition, cgroups are used to control the level of access to resources such as CPU and RAM. Further, the Linux kernel calls that a container can be controlled through Linux capabilities and seccomp. One of the fundamental concepts in information security is the principle of least privilege, first articulated as:

> "Every program and privileged user of the system should operate using the least amount of privilege necessary to complete the job."
>
> — Jerome Saltzer

With reference to containers, this means that each container should run with the minimal set of privileges possible for its effective operation. Applying this principle makes an attacker's life much harder; even if a vulnerability is found in a container, it will be difficult for the attacker to effectively exploit the weakness. And if a container cannot access a vulnerable feature, it cannot be exploited.

A large and easy win for security is to run containers with read-only filesystems. In Docker, this is achieved by simply passing the --read-only flag to docker run. With this in place, any attacker that exploits a vulnerability will find it much harder to manipulate the system; they will be unable to write malicious scripts to the filesystem or to modify the contents of files. Many applications will want to write out to file, but this can be accommodated by using tmpfs or volumes for specific files or directories.

Constraining access to other resources can also be effective. Limiting the amount of memory available to a container will prevent attackers from consuming all the memory on the host and starving out other running services. Limiting CPU and network bandwidth can prevent attackers from running resource-heavy processes such as Bitcoin mining or torrent peers.

Perhaps the most common mistake when running containers in production is having containers which run as the root user. While building an image, root privileges are typically required in order to install software and configure the image. However, the main process that is executed when the container starts should not run as root. If it does, any attacker who compromises the process will have root-level privileges inside the container. Much worse, as users are not namespaced by default, should the attacker manage to break out of the container and onto the host, they might be able to get full root-level privileges on the host.

To prevent this, always ensure Dockerfiles declare a non-privileged user and switch to it before executing the main process. Since Docker 1.10, there has been optional support for enabling user namespacing, which automatically maps the user in a container to a high-numbered user on the host. This works, but currently has several drawbacks, including problems using read-only filesystems and volumes. Many of these problems are being resolved upstream in the Linux community at publication time, so expect that user namespace support will become more viable for a larger percentage of use cases in the near future.

Limiting the kernel calls a container can make also significantly reduces the attack surface, both by constraining what an attacker can do and reducing exposure to vulnerabilities in the kernel code. The primary mechanism for limiting privileges is using Linux capabilities. Linux defines around 40 capabilities, which map onto sets of kernel calls. Container runtimes, including rkt and Docker, allow the user to select which privileges a container should run with. These capabilities map onto around 330 system calls, which means several capabilities, notably SYS_ADMIN, map onto a large number of calls. For even finer control over which kernel calls are allowed, Docker now has seccomp support for specifying exactly which calls can be used, and ships with a default seccomp policy that has already shown to be effective at mitigating problems in the Linux kernel. The main problem with both approaches is figuring out the minimal set of kernel calls your application needs to make. Simply running with different levels of capabilities and checking for failures is effective but time-consuming, and may miss problems in untested parts of code.

Potentially, existing tools can be helpful to determine your application's use of syscalls without resorting to trial and error. If you are able to fully exercise your application's code paths with tracing capabilities, like

strace2elastic, this will provide a report of used syscalls within your application during the container's runtime.

While OS-level isolation and the enforcement of least privilege is critical, isolation also needs to be tied to application logic. Without understanding of the application that is running on the host, OS-level isolation may not be in itself effective.

Runtime Threat Detection and Response

No matter how good a job you do with vulnerability scanning and container hardening, there are always unknown bugs and vulnerabilities that may manifest in the runtime and cause intrusions or compromises. That is why it's important to outfit your system with real-time threat detection and incident response capabilities.

Containerized applications, compared to their monolithic counterparts, are distinctly more minimal and immutable. This makes it possible to derive a baseline for your application that is of a higher fidelity than with traditional, monolithic applications. Using this baseline, you should be able to detect real-time threats, anomalies, and active compromises, with a lower false-positive rate than seen with traditional anomaly detection.

Behavior baselining, where a security mechanism focuses on understanding an application or system's typical behavior in order to identify anomalies, was one of the hottest trends at Blackhat 2016. The key to behavior baselining is to automate — as much as you can — the derivation of the baseline, the continuous monitoring, and the detection and response. Today, most organizations accomplish behavior baselining with a combination of manual labor and data science. However, due to the transient nature of containers, it is especially important that the whole process be automated.

Active response goes hand-in-hand with baselining. Active response is how to respond to an attack, a compromise or an anomaly as soon as it is detected. The response can come in many different forms, such as alerting responsible personnel, communicating with enterprise ticketing systems, or applying some pre-determined corrective actions to the system and the application.

In the container environment, an active response could mean performing additional logging, applying additional isolation rules, disabling a user dynamically, or even actively deleting the container. Again, automation is key here — all actions performed must not interfere with application logic in a negative way, such as getting the system in an inconsistent state or interfering with non-idempotent operations.

Some of the products currently offering this level of runtime threat detection and response include Aqua Security, Joyent Triton SmartOS and Twistlock. As more mission-critical applications move to containers, automating runtime threat detection and response will be increasingly important to container security. The ability to correlate information, analyze indicators of compromise, and manage forensics and response actions, in an automated fashion, will be the only way to scale up runtime security for a containerized world.

Access Controls

The Linux kernel has support for security modules that can apply policies prior to the execution of kernel calls. The two most common security modules are AppArmor and SELinux, both of which implement what is known as mandatory access control (MAC). MAC will check that a user or process has the rights to perform various actions, such as reading and writing, on an object such as a file, socket or process. The access policy is defined centrally and cannot be changed by users. This contrasts with the

standard Unix model of files and permissions, which can be changed by users with sufficient privileges at any time, sometimes known as discretionary access control or DAC.

SELinux was originally developed by the National Security Agency (NSA), but is now largely developed by Red Hat and found in their distributions. While using SELinux does add a significant layer of extra security, it can be somewhat difficult to use. Upon enabling SELinux, the first thing you will notice is that volumes don't work as expected and extra flags are needed to control their labels. AppArmor is similar, but less comprehensive than SELinux, and doesn't have the same control over volumes. It is enabled by default on Debian and Ubuntu distributions.

In both cases, it is possible to create special policies for running particular containers; e.g., a web server policy for running Apache or NGINX that allows certain network operations but disallows various other calls. Ideally, all images would have their own specially crafted policy, but creating such policies tends to be a frustrating exercise, eased slightly by third party utilities such as bane. In the future, we can expect to see an integrated security profile that travels with containers, specifying settings for kernel calls, SELinux/AppArmor profiles and resource requirements.

Further to the topic of access control, it's important to note that anyone with the rights to run Docker containers effectively has root privileges on that host — they can mount and edit any file, or create setuid (set user ID upon execution) binaries that can be copied back to the host. In most situations, this is just something to be aware of, but some organizations will want finer-grained control over user rights. To this end, organizations may want to look at using higher-level platforms such as Docker Datacenter and OpenShift, or tooling, such as Aqua Security and Twistlock, to add such controls.

Conclusion

It is essential for organizations to consider security when implementing a container-based workflow or running containers in production. Security affects the entire workflow and needs to be considered from the start. Image provenance starts with the developers building, pulling and pushing images on their laptops, continues through the CI and testing phases, and ends with the containers running in production.

Containers and the golden image approach enable new ways of working and tooling, especially around image scanning and auditing. Organizations are better able to keep track of the software running in production and can much more easily and quickly react to vulnerabilities. Updated base images can be tested, integrated and deployed in minutes. Image signing validates the authenticity of containers and ensures that attackers have not tampered with their contents.

The future will bring more important features, as verified and signed images become common and features, such as integrated security profiles, are added. In the coming months, the security benefits alone will be a strong reason for organizations to make the move to containers.

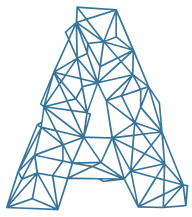
A HISTORY OF SECURITY IN CONTAINER ADOPTION

In this discussion with Bryan Cantrill of Joyent, we delve into a discussion surrounding the security of containers at scale, how hardware visualization impacts container security, and the ways in which Joyent has contributed to the ongoing use of containers in production. Cantrill mentions that the security issue for them is especially focused on multi-tenant security with containers. You can remove the problems around hardware virtualization entirely by deploying multi-tenant environments on bare metal in Triton. It's not just a matter of what's old is new again in the security world, but a real and qualitative increase in the complexity and velocity of systems. **Listen on SoundCloud** or **Listen on YouTube**

***Bryan Cantrill** is the chief technical officer at Joyent, where he oversees worldwide development of the Triton Elastic Container Service, as well as SmartOS, SmartDataCenter and Node.js platforms. Prior to joining Joyent, Bryan served as a distinguished engineer at Sun Microsystems, where he spent over a decade working on system software. In particular, he co-designed and implemented DTrace, a facility for dynamic instrumentation of production systems. Bryan also cofounded the Fishworks group at Sun, where he designed and implemented the DTrace-based analytics facility for the Sun Storage 7000 series of appliances.*

METHODS FOR DEALING WITH CONTAINER STORAGE

by **JANAKIRAM MSV**



few years ago, when virtualization was introduced to IT administrators, there was an attempt to standardize the virtual machine (VM) as the unit of deployment. Each new build and version resulted in a new VM template. Eventually, developers and administrators started creating new images and provisioning VMs, which resulted in a phenomenon called "VM sprawl." Each template and the provisioned VM occupied a few gigabytes of storage, which led to inefficient storage management. Given the large size of VM images, organizations realized that it wasn't practical to create a new VM template for each version.

One of the goals for Docker was to avoid the pitfall illustrated by "VM sprawl." The only way Docker could avoid the trap of fragmented images was to adopt a different storage mechanism for its images and containers.

Another goal that was critical to Docker was the separation of filesystems to create isolation between the host system and containers. This isolation was core to the security of containerized applications. To meet these requirements, Docker adopted a union filesystem architecture for the images and containers.

Union filesystems represent a logical filesystem by grouping different directories and filesystems together. Each filesystem is made available as a branch, which becomes a separate layer. Docker images are based on a union filesystem, where each branch represents a new layer. It allows images to be constructed and deconstructed as needed instead of creating a large, monolithic image.

Docker's use of existing Linux union filesystems is ideal for running applications that can rapidly scale. Since they are self-contained, Docker containers can be launched on any host with no dependencies or affinity to a specific host. While this is certainly an advantage for web-scale workloads, it becomes a challenge to run stateful applications that deal with persistent data. Workloads, such as relational databases, NoSQL databases, content management systems, and big data stacks, demand persistence and durability of data. Some workloads, like content management systems, also require shared data access across multiple application instances.

When a Docker image is pulled from the registry, the engine downloads all the dependent layers to the host. When a container is launched from a downloaded image comprised of many layers, Docker uses the copy-on-write capabilities of the available union filesystem to add a writeable "working directory" — or temporary filesystem — on top of the existing read-only layers. When Docker first starts a container, this initial read-write layer is empty until changes are made to the filesystem by the running container process. When a Docker image is created from an existing container, only the changes made — which have all been "copied up" to this writeable working directory — are added into the new layer. This approach enables reuse of images without duplication or fragmentation.

When a process attempts to write to an existing file, the filesystem implementing the copy-on-write feature creates a copy of the file in the

topmost working layer. All other processes using the original image's layers will continue to access the read-only, original version of the layer. This technique optimizes both image disk space usage and the performance of container start times.

Strategies to Manage Persistent Data

Docker's layered storage implementation is designed for portability, efficiency and performance. It is optimized for storing, retrieving, and transferring images across different environments. When a container is deleted, all of the data written to the container is deleted along with it.

As a best practice, it is recommended to isolate the data from a container to retain the benefits of adopting containerization. Data management should be distinctly separate from the container lifecycle. There are multiple strategies to add persistence to containers. We will evaluate the options that are available out-of-the box with Docker, followed by the scenarios that are enabled by the ecosystem.

Host-Based Persistence

Host-based persistence is one of the early implementations of data durability in containers, which has matured to support multiple use cases. In this architecture, containers depend on the underlying host for persistence and storage. This option bypasses the specific union filesystem backends to expose the native filesystem of the host. Data stored within the directory is visible inside the container mount namespace. The data is persisted outside of the container, which means it will be available when a container is removed.

In host-based persistence, multiple containers can share one or more volumes. In a scenario where multiple containers are writing to a single shared volume, it can cause data corruption. Developers need to ensure

that the applications are designed to write to shared data stores.

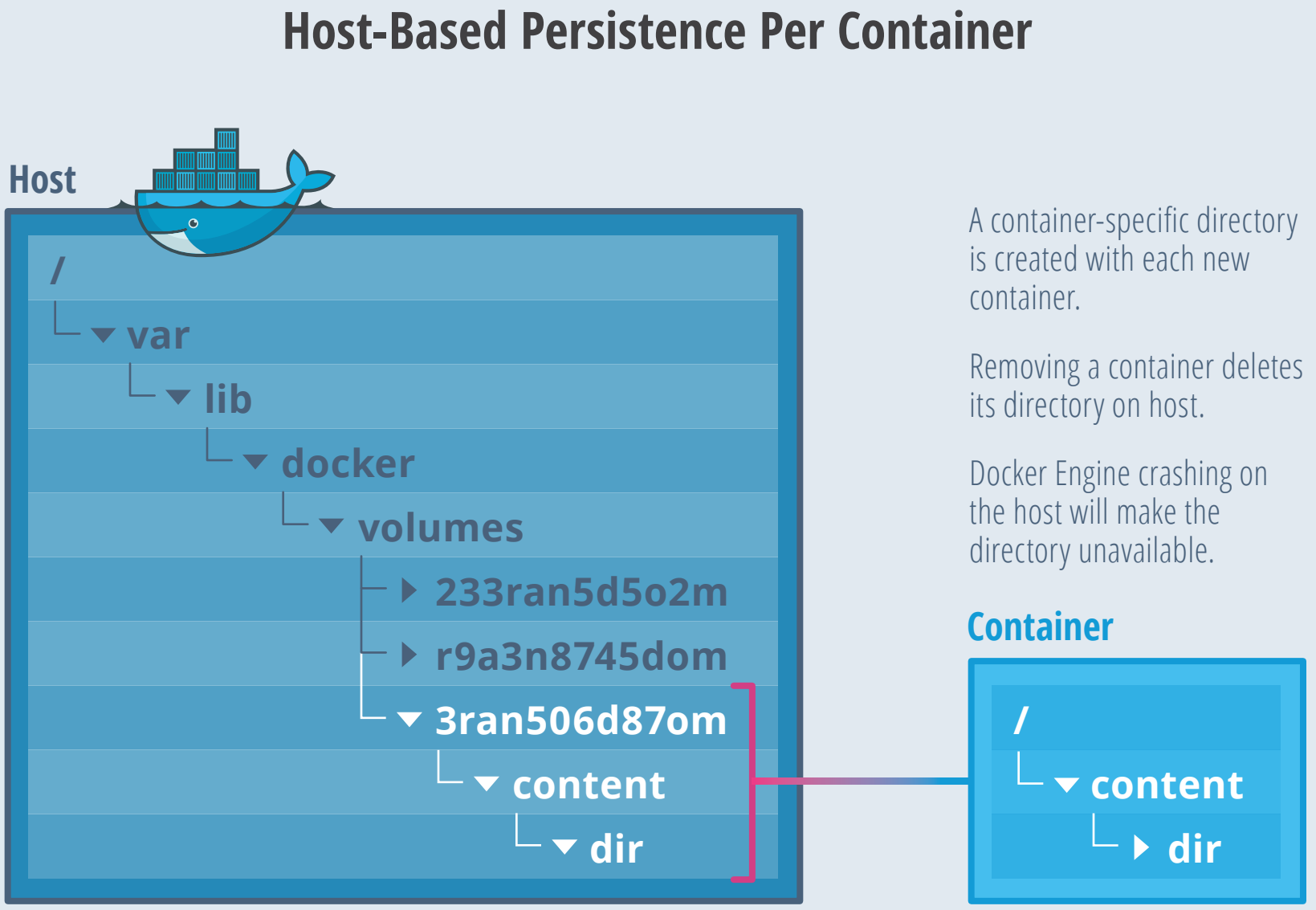
Data volumes are directly accessible from the Docker host. This means you can read and write to them with normal Linux tools. In most cases you should not do this, as it can cause data corruption if your containers and applications are unaware of your direct access.

There are three ways of using host-based persistence, with subtle differences in the way they are implemented.

Implicit Per-Container Storage

The first mechanism will create an implicit storage sandbox for the container that requested host-based persistence. The directory is created by default at /var/lib/docker/volumes on the host during the creation of the container. When the running container is removed, the directory is

FIG 1: *Data storage using the container's host relies too much on stability, and otherwise prevents container portability.*



Source: Janakiram MSV

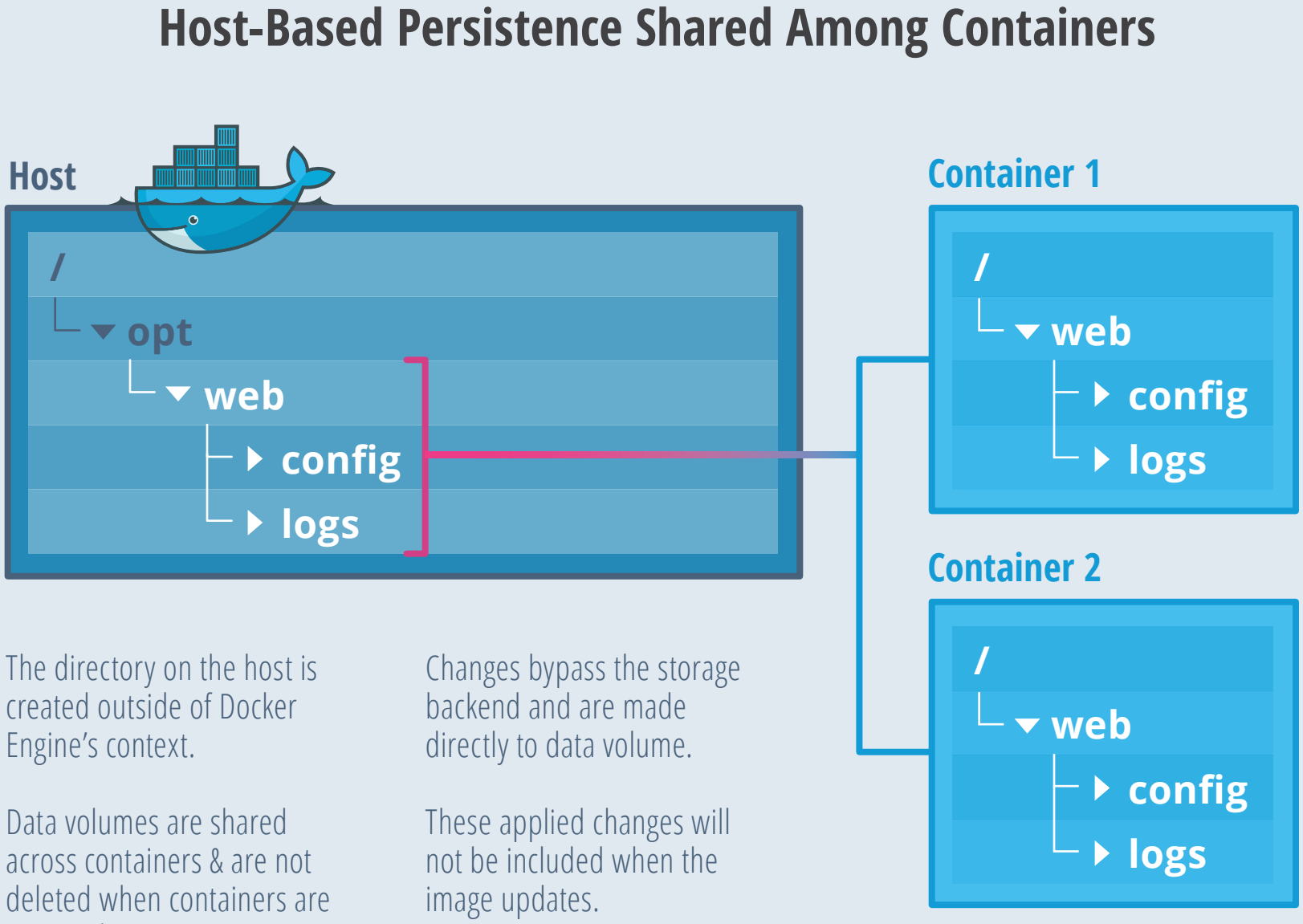
automatically deleted on the host by the Docker Engine. The directory may also become unavailable if the Docker Engine crashes on the host. The key thing to understand is that the data stored in the sandbox is not available to other containers, except the one that requested it.

Explicit Shared Storage (Data Volumes)

We can choose the second technique if there is a need to share data across multiple containers running on the same host. In this scenario, an explicit location on the host filesystem is exposed as a mount within one or more containers.

This becomes especially useful when multiple containers need read-write access to the same directory. For example, containers running an Apache web server can centrally store logs to the same directory, making it easier to process the logs.

FIG 2: *Shared host persistence, called data volumes, are available even if the container is deleted.*



Source: Janakiram MSV

Since the directory on the host is created outside of Docker Engine's context, it is available even after removing every container or even stopping Docker Engine. Since this shared mount point is fully outside the control of Docker Engine's storage backend, it is not part of the layered, union filesystem approach.

This technique is the most popular one used by DevOps teams. Referred to as data volumes in Docker, it offers the following benefits:

- Data volumes can be shared and reused across multiple containers.
- Changes made to a data volume are made directly, bypassing the engine's storage backend image layers implementation.
- Changes applied to a data volume will not be included when the image gets updated.
- Data volumes are available even if the container itself is deleted.

Shared Multi-Host Storage

While both techniques discussed above offer varying levels of persistence and durability, there is one major drawback with them — they make the containers nonportable. The data residing on the host will not move along with the container, which creates a tight bind between the host and container.

Customers deploying containerized workloads in production often run them in a clustered environment, where multiple hosts participate to deliver required compute, network and storage capabilities. This scenario demands distributed storage that is made available to all hosts, and is then exposed to the containers through a consistent namespace.

Shared filesystems, such as Ceph, GlusterFS, Network File System (NFS) and others, can be used to configure a distributed filesystem on each host

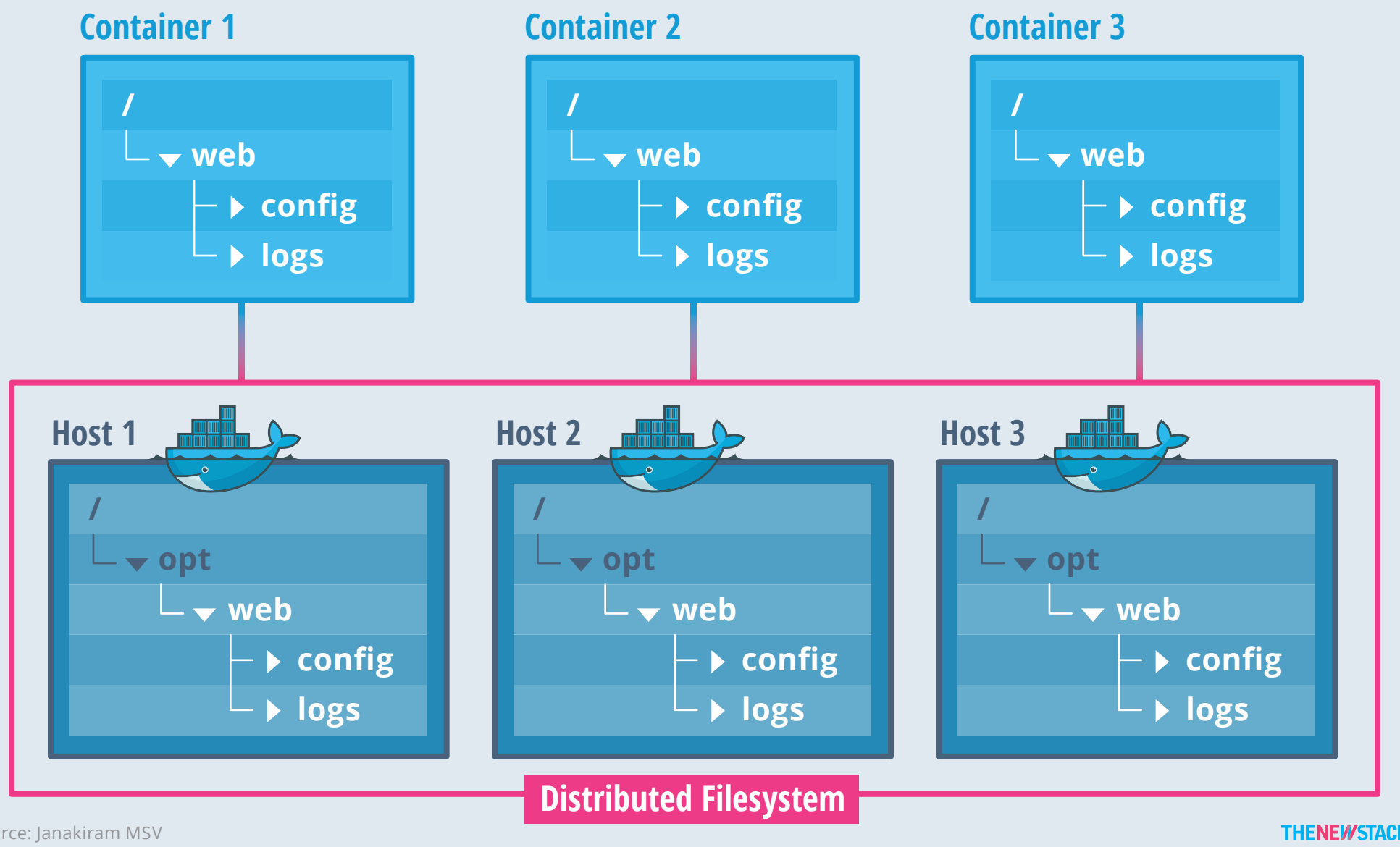


FIG 3: *Multi-host persistence allows for data storage that takes advantage of the portability of containers.*

running Docker containers. By creating a consistent naming convention and unified namespace, all running containers will have access to the underlying durable storage backend, irrespective of the host from which they are deployed.

Shared multi-host storage takes advantage of a distributed filesystem combined with the explicit storage technique. Since the mount point is available on all nodes, it can be leveraged to create a shared mount point among containers.

Containerized workloads running on orchestration engines deployed in production environments can configure a distributed filesystem on a subset of cluster nodes. These nodes will be designated for scheduling containers that need long-term durability and persistence.

Orchestration engines provide a mechanism to specify hosts during the scheduling of containers. Docker Swarm filters come with container configuration filters, which define the nodes to use when creating and running containers. In Kubernetes, labels can be used to target a set of nodes when deploying pods. Kubernetes also utilizes Pet Sets, a group of stateful pods that have a stronger requirement for identity.

Typical Operations Supported By Host-Based Persistence

Thanks to tight integration with the core container engine, host-based persistence is the simplest to configure. Development or operations teams perform the following tasks to enable host-based persistence — we are considering explicit shared storage and shared multi-host storage scenarios for this workflow:

- **Create volumes:** This is the first step for enabling persistence in containers. It results in the Docker Engine creating a designated volume which points to the host filesystem.
- **Launching stateful containers:** The persistent volumes are then associated with one or more containers during launch time.
- **Backing up data:** Data stored in volumes can be easily backed up to a tar or zip file. Refer to Docker documentation for guidance on this process.
- **Migrating and restoring data:** The backed up data can then be migrated or restored on a different host by creating a new data volume and decompressing the file.
- **Deleting volumes:** Data volumes are not automatically deleted after removing associated containers; they will need to be manually deleted by the operations team.

- **Configuring a distributed filesystem (optional):** In multi-host scenarios, IT may have to configure a shared filesystem, spanning multiple physical or virtual servers.

Top Use Cases for Using Host-based Persistence

Host-based persistence may be considered for the following scenarios:

- **Databases:** It can be faster to write to a volume than the copy-on-write layer. This is applicable when running relational and NoSQL databases.

- **Host-mounting source code:** In a development environment where source code needs to be shared between the host and containers, host-based persistence comes in handy. Since the container accesses the same version as the host, it's easy to debug and test in a container environment. Developers work in their normal IDE, editing files on their local Docker host, and those changes are reflected immediately inside the container.

- **Master-Worker:** In a scenario where data needs to be shared with two containers acting as master and worker, host-based persistence should be used. For example, the data aggregated by the master container is processed by a worker container.

Volume Plugins

Although host-based persistence is a valuable addition to Docker for specific use cases, it has the significant drawback of limiting the portability of containers to a specific host. It also doesn't take advantage of specialized storage backends optimized for data-intensive workloads. To solve these limitations, volume plugins have been added to Docker to extend the capabilities of containers to a variety of storage backends,

without forcing changes to the application design or deployment architecture.

Starting with version 1.8, Docker introduced support for third-party volume plugins. Existing tools, including Docker command-line interface (CLI), Compose and Swarm, work seamlessly with plugins. Developers can even create custom plugins based on Docker's specifications and guidance.

According to Docker, volume plugins enable engine deployments to be integrated with external storage systems and data volumes to persist beyond the lifetime of a single engine host. Customers can start with the default local driver that ships along with Docker, and move to a third-party plugin to meets specific storage requirements. Volume plugins also enable containerized applications to interface with filesystems, object storage, block storage, and software-defined storage.

As of June 2016, Docker supports over a dozen third-party volume plugins for use with Azure File Storage, Google Compute Engine persistent disks, NetApp Storage and vSphere. In addition, projects like Rancher Convoy can provide access to multiple backends at the same time.

Basics of Volume Plugin Architecture

Docker ships with a default driver that supports local, host-based volumes. When additional plugins are available, the same workflow can be extended to support new backends. This architecture is based on Docker's philosophy of "batteries included, but replaceable." The third-party volume plugins are installed separately, which typically ship with their own command line tools to manage the lifecycle of storage volumes.

Docker's volume plugins can support multiple backend drivers that interface with popular filesystems, block storage devices, object storage services and distributed filesystems storage.

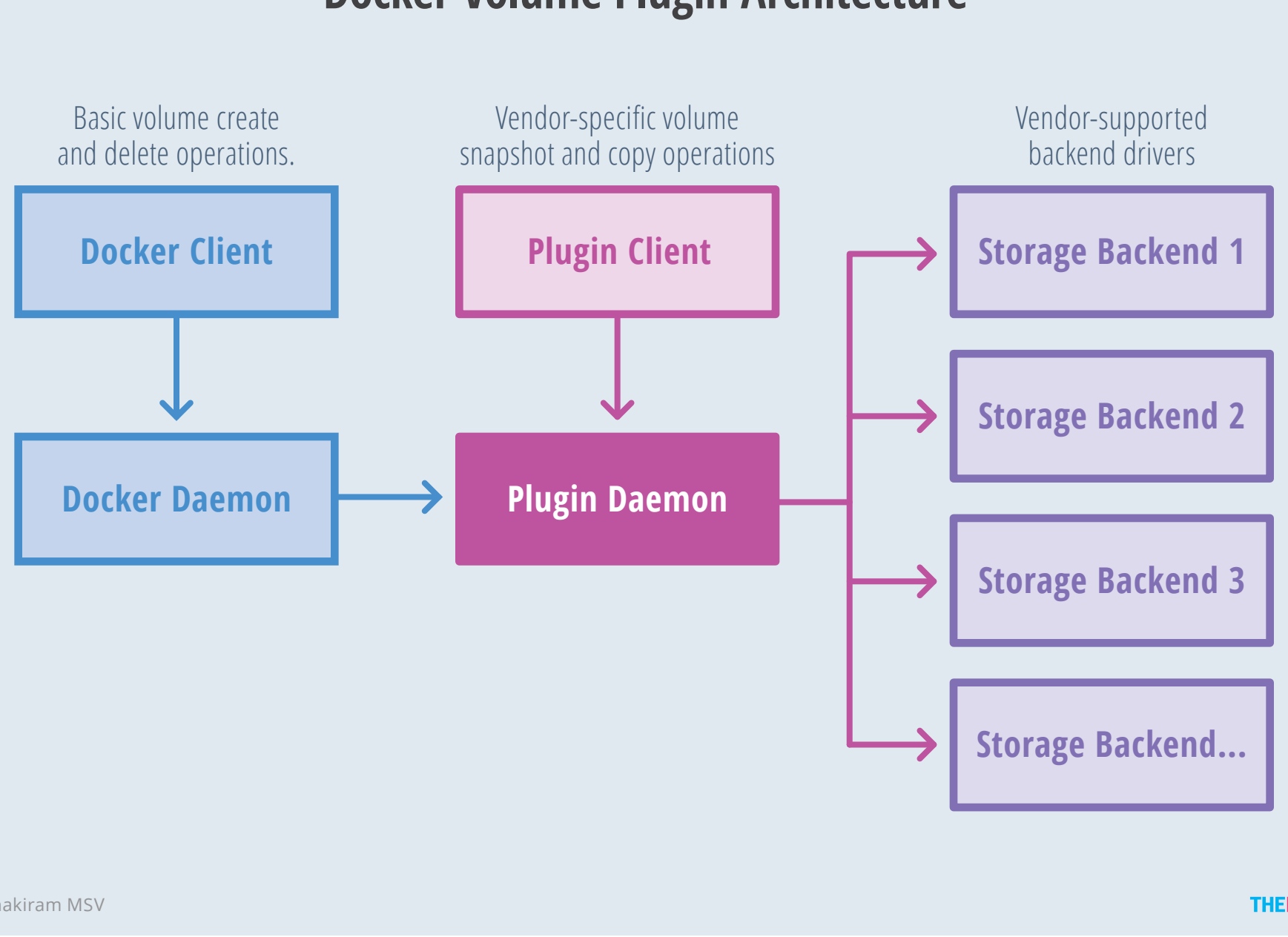


FIG 4: *Third-party volume plugins utilize the same workflow available to Docker drivers.*

Typical Operations Supported By Volume Plugins

Volume plugins typically install a daemon responsible for managing the interaction with storage backends. A client in the form of a command-line interface (CLI) talks to the daemon to perform storage-specific tasks on the volume. The operations supported by the CLI go beyond the standard tasks that the Docker CLI can perform.

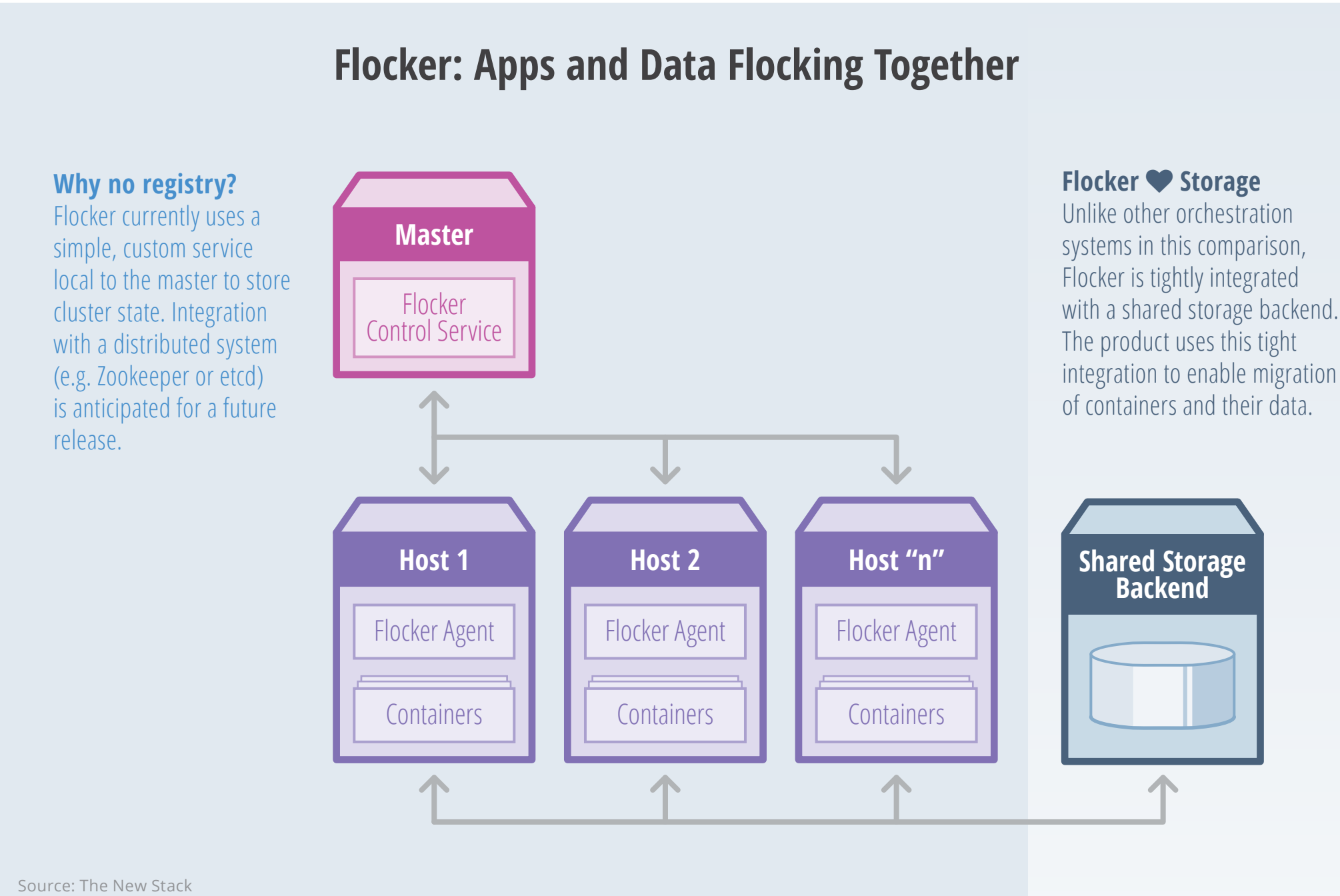
The volume plugin clients enable the following tasks as part of the lifecycle management:

- **Creating provisioned volumes:** This step involves creating devices that can be accessed while creating a volume from a standard Docker CLI.

- **Taking snapshot of volumes:** Many plugins support creating point-in-time snapshots of volumes. These incremental snapshots only contain the delta of changes made since the last snapshot, thus maintaining a small size.
- **Backing up snapshots to external sources:** Optionally, volume plugin tools support backing up snapshots to sources such as Amazon S3 and Azure Storage.
- **Restoring volumes on any supported host:** Plugins enable easy migration of data from one host to another by restoring the backups and snapshots.

Flocker from ClusterHQ is one of the first volume plugins to integrate with Docker. The Flocker data volume, called a dataset, is portable and can be

FIG 5: *Flocker utilizes a portable volume architecture that can move between hosts in a cluster.*



used with any container within the cluster. It manages Docker containers and data volumes together, enabling the volumes to follow the containers when they move between different hosts in the cluster.

Flocker works with mainstream orchestration engines such as Docker Swarm, Kubernetes and Mesos. It supports storage environments ranging from Amazon Elastic Block Store (EBS), GCE persistent disk, OpenStack Cinder, vSAN, vSphere and more.

There are many open source volume plugins to support a variety of storage backends. Please refer to Docker's plugin page for the latest list of available plugins.

Top Use Cases for Volume Plugins

Volume plugins target scenarios typically used in production environments. The following list highlights different use cases:

- **Data-intensive applications:** Since volume plugins have drivers for specialized storage backends, they can deliver the required performance demanded by data-intensive workloads such as big data processing and video transcoding.
- **Database migration:** Volume plugins make it easy to move data across hosts in the form of snapshots, which enable migration of production databases from one host to another with minimum downtime. Through this, containers in production environments can be migrated to powerful hosts or virtual machines.
- **Stateful application failover:** Using volume plugins with a supported shared storage backend like Amazon EBS, customers can manually failover containers to a new machine and reattach an existing data volume. This enables transparent failover of stateful applications.

- **Reduced Mean Time Between Failures (MTBF):** With volume plugins connected via a shared storage backend, operations teams can speed up cluster time-to-recovery by attaching a new database container to an existing data volume. This results in faster recovery of failed systems.

In the next section, we will take a closer look at different container storage choices made available by this vibrant ecosystem.

Container Storage Ecosystem

Since storage is a key building block of the container infrastructure, many ecosystem players have started to focus on building container-specific storage offerings.

The container storage ecosystem can be broadly classified into software-defined storage providers, specialized appliances providers and block storage providers. While there are a few dozen entities delivering storage solutions from the container ecosystem, we will explore some of the prominent players from each category.

Software-Defined Storage Providers

The rise of containers in enterprise has led to the creation of a new class of storage optimized for containerized workloads. Existing storage technologies, such as network-attached storage (NAS) and storage area network (SAN), are not designed to run containerized applications. Software-defined storage abstracts these traditional types of storage to expose the virtual disks to the more modern applications.

Container-defined storage, a new breed of storage, is a logical evolution of software-defined storage, which is purpose-built to match the simplicity, performance and speed of containers. Container-defined storage runs on commodity hardware, featuring a scale-out block storage, which in itself is

deployed as a container. It provides per-container storage, distributed file access, unified global namespace, fine-grained access control, and a tight integration with the cluster management software. Many providers make money by selling these services on top of commodities or bundled with a cloud provider's offering.

One of the key advantages of using software-defined storage for containers is the ability to virtualize storage, which may be based on faster solid-state drives (SSD) or magnetic disks. Aggregating disparate storage enables IT to utilize existing storage investments. Some flavors of container-defined storage can automatically place I/O-intensive datasets on faster SSDs while moving the archival data to magnetic disks. This delivers the right level of performance for workloads, such as online transaction processing (OLTP), which demand high input/output operations per second (IOPS).

Many companies are working on integrating software-defined storage with containers, with many of them selling appliances or Storage as a Service. Portworx, Hedvig, CoreOS Torus, EMC libStorage, Joyent Manta and Blockbridge all provide developers with access to their software without requiring them to buy something else. StorageOS, Robin Systems and Quobyte are examples of companies that do not provide unbundled access to their software.

Storage Appliance Providers

The virtualization of compute, storage and networking led to the evolution of software-defined infrastructure. Vendors like Dell, VCE and Nutanix started to ship appliances that delivered datacenters in a box. These appliances came with bundled hypervisor, storage and networking capabilities, along with management software to orchestrate the virtual infrastructure.

With containers becoming a popular choice, some startups are building appliances that deliver end-to-end infrastructure for containerized workloads. They have purpose-built converged infrastructure for containers that comes with network interoperability and persistent storage.

Robin Systems and Portworx are two of several software-defined storage providers that sell software appliances. Diamanti is an early mover in the space of container-based converged infrastructure. Its appliance comes loaded with industry-standard software — Linux, Docker and choice of orchestration engine — but it's unique because it provides container networking at the hardware level. Other companies, such as Datera, offer storage appliances as solutions for container use cases.

As containers become mainstream in the enterprise, we can expect to see the rise of converged infrastructure for containerized workloads. Containers solve the problem of assembling the right technology stack for cloud-native applications and microservices.

Object and Block Storage Providers

Key benefits of running containers are realized when they are deployed in web-scale environments. Customers leveraging public clouds often rely on object storage services, such as Amazon S3, IBM Bluemix Object Storage and Joyent Manta, as well as block storage devices such as Elastic Block Storage (EBS) or Google Compute Engine (GCE) persistent disks. To enable easy integration with the infrastructure, these cloud providers are investing in storage drivers and plugins that bring persistence to containers. DevOps teams can host image registries in the public cloud backed by object storage. Block storage devices deliver performance and durability to workloads.

These investments in storage drivers and plugins by cloud providers are

primarily meant to run hosted container management services or Containers as a Service (CaaS) offerings. Recently, at DockerCon 2016, Docker announced native support for AWS and Azure. This will accelerate the development and optimization of storage drivers for object and block storage.

Summary

Storage is one of the key building blocks of a viable enterprise container infrastructure. Though Docker made it easy to add persistence based on data volumes, the ecosystem is taking it to the next level. Volume plugins are a major step towards integrating containers with some of the latest innovations in the storage industry. Providers are making it possible to tap into the power of enterprise storage platforms.

After revolutionizing the virtualization market, software-defined storage is poised to witness a huge growth with containers. The concept of mixing and matching low-cost magnetic disks with advanced flash and SSD will benefit enterprises running containerized workloads in production, such as IBM FlashSystem arrays being used to power cognitive computing services. This combination can use existing commodity hardware to build storage pools consumed by cloud-native applications.

It's only a matter of time before converged infrastructure embraces containers to deliver turn-key infrastructure platforms optimized to run complex workloads. The compute building block that relies on hypervisors and VMs is gradually shifting to containers. Vendors, like Diamanti, are gearing up to ride the new wave of converged infrastructure powered by containers.

Public cloud providers with robust storage infrastructure are getting ready for containers. Object storage, block storage and shared filesystem

services will get dedicated drivers and plugins to maintain images in private registries and run I/O-intensive containerized workloads in the cloud. Containers as a Service will also drive the demand for native drivers for public cloud storage.

SOFTWARE-BASED NETWORKING AND SECURITY

In this discussion with Hari Krishnan and Harmeet Sahni of Nuage Networks, we talk about how to use software-defined networking (SDN) to address the networking and security needs of containers. SDN can help prevent infrastructure-related performance bottlenecks, as well as provide the fine-grained policy enforcement that allows users to handle dynamic container environments. Nuage Network's Virtual Services Platform (VSP) provides these capabilities in all types of cloud environments, featuring user intent-focused definition, virtual routing and switching components, and more. **Listen on SoundCloud** or **Listen on YouTube**

***Hari Krishnan** is senior director of product management for security at Nuage Networks, an SDN venture from Nokia. Krishnan brings over 18 years of experience leading product management, strategy and marketing for network security, SDN, cloud, virtualization, IP networking and management software products to enterprise and service provider markets. He previously held leadership roles at Cisco, VMware, F5 Networks and Nominum.*

***Harmeet Sahni,** director of product management at Nuage Networks, leads product management and product strategy for container networking, hybrid cloud and integrations with application delivery partners. Harmeet brings over 15 years of experience leading product management, strategy and development of SDN, cloud application delivery, analytics and network infrastructure software products to enterprise and service provider customers. He previously held leadership roles at Riverbed Technology and Juniper Networks.*

IDENTIFYING AND SOLVING ISSUES IN CONTAINERIZED PRODUCTION ENVIRONMENTS

by **VIVEK JUNEJA**

So you have an application that is composed around containers. You have lightweight base images, a centralized container registry, and integration with the deployment and continuous integration (CI) pipeline — everything needed to get containers working at full scale on your hardware. For running a multitier application, you spent time on using a service discovery mechanism for your application containers. You have a logging mechanism that pulls out the information from each container and ships them to a server to be indexed. Using a monitoring tool that is well suited for this era when machines are disposable, you see an aggregate of your monitoring data, giving you a view of the data grouped around container roles. Everything falls nicely into place.

You're ready to take this to the next level by connecting your pipeline to production. The production environment is where the containers will see the most entropy. Rolling containers into production requires that you spend your time building a canary release system to implement a rolling upgrade process. Every change travels neatly from the development environment to your production environment, shutting down one

container at a time and replacing them with a brand new version of your code. This is what usually comes to mind when we talk about adopting containers at a high level.

However, to the true practitioner, this is the tip of the iceberg. Doing everything mentioned earlier still does not guarantee a perfect environment for your containers. There's still potential to have your plans derailed, and worse, create conditions that may shake your confidence in containers. We'll explore these issues around container networking, storage and security.

Container Networking

Containers do not live in isolation; they need to connect with other services. Containers need to be discoverable and available for connection to other services. Irrespective of their location in a given fleet of machines, the goal is to reliably and quickly reach out to a destination container. Networking in the container realm is often intertwined with service discovery. While networks change across development, testing and production environments, service discovery remains consistent across environments. This means that the service discovery mechanism must remain common across the varied networks where containers are deployed.

If you have just started using containers in production, there are some key questions that need answering to help stabilize your approach:

- How do you select the right network configuration for a given scenario? Should you use a bridge, overlay, underlay or another networking approach?
- How does service discovery integrate with the various container network configurations?

- How do you monitor a container network and identify bottlenecks with its performance?
- How do you visualize a network topology running across multiple hosts?
- How do you secure container networks?
- How do you isolate networks when running containers belonging to varied tenants on the same physical or virtual hosts?

We will address each of these concerns before moving on to other misunderstood aspects of containers in production.

Network Configuration

It is recommended to have the containers use the host network, instead of a bridged network, if the services running in the container need to be exposed to outside users. This is primarily because the bridged network causes latency due to the virtual Ethernet (vEth) network connection. When containers use the host network, port number conflicts could be a cause of concern. To resolve that, the application service in the container is configured to run on a dynamic port provided at runtime, rather than a default port. For example, when running a Tomcat container for a Java application, the server and Apache JServ Protocol (AJP) port numbers could be supplied at runtime using the operating system (OS) environment variables.

```
docker run -d -e SERVER_HTTP_PORT=8080 -e SERVER_
AJP_PORT=8005 tomcat:8
```

The environment (ENV) variables **`SERVER_HTTP_PORT and SERVER_AJP_PORT`** are used as references, since the Tomcat image is modified to run the Tomcat server on the defined server ports. This will prevent the containers from binding to a consistent port on the host, and will allow

multiple instances of containers running the same image, at the same time, on the same host.

The host network also prevents the constant change of iptables, which is common with the bridge network. You would not want to do that change in a production environment where iptables could be used for firewall configuration. The bridged network is commonly used in development and testing environments to allow multiple concurrent containers of the same kind to run on a set of shared hosts. Port mappings are the way to allow the bridged containers to be accessed from end users.

Container orchestration platforms, like Kubernetes, also offer a pod model for container networks that share the same IP address. This is useful for grouping application services in containers that usually work together.

Docker features overlay networks that enable easy creation of multi-host networks with per-container internet protocol (IP) support. Other solutions, like Calico, Flannel and Weave, can integrate with Docker as a network plugin. This space is rapidly developing, so the advice is to test the performance and reliability of these technologies first before adopting them into production.

Service Discovery and Container Networking

Service discovery is usually an infrastructure concern; it allows applications deployed as containers to transparently access each other via means like domain name system (DNS). If the containers are deployed on the host network in production, then a proxy must exist that is able to route incoming requests to the containers. Previously, a service discovery solution based on Consul and Registrator was an easy setup mechanism for discoverable containers.

This has evolved, thanks to the introduction of overlay networks and an

array of third-party plugins like Calico and Weave. Implementation requires use of a key-value store to coordinate network updates across hosts. In some solutions, DNS is used as the basis of the service discovery. However, DNS has its own pitfalls, as local caching may affect the discovery process in cases where containers are frequently changing or moving across hosts. Other solutions include services, like HAProxy or Traefik, that can work as a reverse proxy with different orchestration backends.

A port-based service discovery is often hard to use, but will work for small-scale clusters with minimal applications. Managing the lifecycle of ports while using proxy configurations for discovery could spiral out of control and is difficult to debug. If you choose to integrate a network mode that allows IP per-container, use a service discovery mechanism that is integrated into the network provider. This reduces the number of moving parts in the solution and simplifies relations between networking and service discovery.

Monitoring and Visualizing Container Networks

When dealing with containers in production, it is important to understand how they interact with each other. This is vital to help diagnose issues and alleviate the chance of misconfiguration. Thankfully, the container ecosystem actively supports this requirement. For instance, Weave Scope provides an overview of a containerized application's interconnections across a given set of hosts.

This is crucial to understanding how containers communicate between each other and with other uncontained services. Container-native monitoring tools, like Sysdig, offer a view of real-time traffic movement across container instances. When the container count increases and individual container monitoring becomes difficult, they can be grouped together.

Monitoring and performance management in container networks is the primary topic of the next and final book in The Docker and Container Ecosystem series, Monitoring & Management with Docker and Containers. We'll go more in-depth on this topic then, as the monitoring space itself is complicated and extremely important for showing the value of containers in production.

Isolating Networks

When running multiple services belonging to different tenants on shared infrastructure, isolation needs to be created to protect the network connection between related containers. Docker addresses this need with user-defined networks. This means that containers on the same tenant are connected by a network that is separate from other tenants. Docker also provides overlay networks that span across multiple hosts. There could be any number of overlay networks, with each ideally being used for a related container.

Container Storage

Containers have become an essential component of the continuous delivery (CD) story for most organizations. Moreover, containers are the best packaging means for microservices. Running containers in a production environment entails pulling out layers of filesystem changes and piling them, one over the other, into the container — a pattern commonly used in Docker's popular container runtime.

Some adoptees have their applications bundled as containers, but still rely on local container storage or some form of host-mounted volume. While this works on a small scale, this practice quickly runs out of steam as it scales across tens, hundreds or thousands of hosts. Most container advocates recommend steering away from ephemeral container storage and towards moving state outside of the container.

Lightweight kernels have emerged over the last few years, purpose-built for the demands of applications. This rise is due to the fact that a lightweight container image contributes to a faster deployment, which then leads to rapid feedback loops for developers. Teams running containers in production environments often find themselves doing general housekeeping, which entails getting rid of old container volumes from continuous deployments of new image versions. Regular clean-up activity ensures the hosts never run out of storage space and optimizes the filesystem drivers.

Similar to networking, we want to go over some key questions that need answering before adopting storage for containers:

- How do you select the right filesystem driver for a deployment case?
- How do you select the right persistent storage backend for containers?
- How do you reduce the size of the container images?
- How do you retire old containers to keep the filesystem under control?

Addressing the Filesystem Drivers

Many drivers are available for use with Docker. Advanced multi-layered unification filesystem (AUFS) is very common; it is stable and has had success in deployments. AUFS mounts are very fast, and for small file sizes it offers native read and write speeds. However, it can cause latency with large write operations, primarily because of its design. For large files, it is better to use a data volume from Docker.

OverlayFS is a relatively new driver, and offers faster read and write speeds compared to AUFS; however, it is important to have this tested for a certain period of time before moving to production, primarily due to its new existence in the Linux kernel. Docker offers OverlayFS as a driver, and the container runtime rkt also uses it in the new Linux kernels.

If you choose to use Device Mapper, and are running multiple containers on a host, it is preferred to use real block devices for data and metadata.

Persistent Storage for Containers

The ideal case for a containerized application is to have its state managed completely outside the realms of its execution; this means the container is leveraging an external data source for all of its state requirements. However, you can run stateful services as containers. In that case, you could either choose between having a container-managed volume or mounting a directory from the host to the containers. You could also use a Docker volume plugin, like Flocker, thus allowing the containers to access shared storage. Utilizing plugins makes it much easier for containers to use multiple hosts while still accessing a persistent block storage.

It is also possible to build data-only containers. However, that practice is now discouraged for production environments because named volumes provide better volume management as well as the ability to use other drivers for volume storage. Data-only containers can create problems when identifying the right data container to use, which becomes problematic if you have multiple data containers in your production environment.

One of the issues with host-managed volumes is permission conflict. Having files on the host with different ownership privileges can create issues when containerized applications access them. Then you need to change the ownership of the host-managed files to match the operating user inside the container. This could be painful to manage if the shared host volume is changed by adding new files from a different source.

A better practice is to check how often the persistent data store changes outside the context of the containerized application. If the persistent store is purely managed by the container, meaning all files are created and

changed by the containerized application, then it's prudent to use named volumes. If the change to the persistent data store is also made outside the container, then it's better to use a data volume that uses host-mounted storage. In this case, conflicting file permissions could pose challenges. Therefore, it is required to change the file permissions before the application inside the container accesses them.

Size of Container Images

The general rule of thumb is to keep the container image size as minimal as possible. This helps to reduce the amount of data the container host has to fetch when pulling the image from the repository. One way to alleviate the problem of overgrown and bloated container images is to avoid installing unnecessary packages, especially through the package manager built inside the container. For container runtimes that use copy-on-write (COW) mechanisms, this discipline in creating Dockerfiles also helps create lightweight images. Avoid unnecessary layers — this increases the container image size — by combining run commands into one line, and making them available as one layer when building container images.

When using lightweight base images like Alpine for Docker, you need to make sure they comply with apk package manager, which may have different package names from the ones available through popular package managers like Apt.

Retiring Old Container Images, Volumes and Instances

One common issue when deploying containers is running out of disk space while running different containers and their associated volumes. Most host housekeeping mechanisms need to watch out for old container images that have been around for a while and do not need to be on the host. If your continuous delivery pipeline has been active, putting out tens

or hundreds of changes a day in production, it is likely the old images are piling up. Deleting the unused container images is one tactic available through container runtime tooling. A container runtime like Docker prevents deletion of container volumes when deleting the container instances.

However, it is often the cleanup process that creates the most confusion. Data containers behave like ordinary containers and can be mistakenly flagged for deletion while performing cleaning activities. Moving to named volume containers prevents mix-up with the ordinary application containers. As for the actual named volumes, the removal of the container will not affect the named volumes, and thus the named volumes can be reused across the lifecycle of multiple containers.

Container Security

When containers first started becoming popular, one of the main concerns was whether containers were secure. This concern became more prominent when considering whether to use containers on a shared host running varied tenants. With the granular permission models now available in container runtimes, and with the isolated user, network and process namespaces, the state of container security has greatly stabilized. There are still some areas of debate and concern, however. Allowing containers to use secrets like credentials or access keys is still in contention.

Security concerns also impact the way container storage and networks are implemented. Vulnerability scans and signed container images are becoming well-known practices amongst developers. Container marketplaces like Docker's Hub and Store, have already implemented some of these practices, which benefit users who may have limited or no understanding of these problems.

Here are some key questions that need to be answered when considering the security of container deployments in production environments:

- How do you propagate critical security patches to container images in the production environment?
- How do you validate container images obtained from external sources?
- How do you enable verification of the container images before deploying them in production?
- How do you allow containers to have access to secure keys and credentials without exposing them to the host or other colocated containers?
- How do you prevent a compromised container from disturbing the host or other colocated containers?
- How do you test for secure container deployments?

Integrating Security Patch to Container Base Images

Containerized applications are composed of a base image and the application level changes, which are then baked into the final image for deployment. Usually, the base image is the one that remains less prone to changes, while the application changes are constantly baked in through continuous integration. When security changes are proposed in the form of patches to the operating environment, the change is passed on to the base image.

The base image is rebuilt with the planned change, and is then used as a new base image to rebuild the application containers. It is important to have a consistent image across all environments. A change in the base

image is like any other change and is passed to the production environment using the continuous delivery pipeline.

Trusted Container Images and Deployment

Docker introduced Content Trust from version 1.8 onwards. It uses the developer or publisher's private key to sign the image before pushing the image to the registry. When fetching the same image, the public key of the developer is used for verification. This mechanism can be integrated in the continuous integration and deployment pipeline. Docker also offers an enterprise-grade image storage solution called Docker Trusted Registry. rkt also offers trusted builds and deployment, and can be configured to obtain the keys either from the local store or a remote metadata store.

Access Secrets from Containers

Secrets include any information that you would not want to leak out to unauthorized agents, but are required by the container to access external resources. Common ways to inject these secrets has been to use environment variables or to bundle them in a build manifest.

Both approaches have inherent flaws that can expose the secrets to unauthorized agents. Deploying containers on AWS that need secret keys can take advantage of the identity and access management (IAM) roles instead of passing keys insecurely. The EJSON library from Shopify allows you to encrypt secret information, which can be committed to the source code repository. Keywhiz from Square and Vault from HashiCorp also act as key-value stores.

Separating Compromised Containers

A compromised container can potentially disarm the container host. One way to address this is to enable SELinux and SVirt. SVirt works with Docker to disallow container processes from getting access to the host system. A detailed account has been shared by Project Atomic.

CoreOS has also integrated SVirt with the rkt runtime; it is available by default whenever a new container is run. This is especially critical if the developer uses an untrusted container image from the public web without inspection. Alternatively, for Docker, the flag --cap-drop allows permissions to be revoked from a given container.

Secure Container Deployments

The Docker Bench program provides an automated tool that checks against all the best practices as advocated in the Center for Internet Security (CIS) Benchmark v1.11 report. This will help identify hotspots in your infrastructure that need attention before putting services into production. Another important step is to have a secure container host. The Docker security deployment guidelines is a good resource to strengthen your Docker container runtime environment.

Summary

Successfully adopting containers is a difficult task for many. Adopting the use of containers in production environments is not possible without giving an enormous amount of consideration to container networking, storage and security. As more organizations practice this art, new patterns and practices will emerge, making container deployments the de facto for any applications built in the future. The goal is to rethink how security, storage and networking will evolve when container count increases in production, running not just tens or hundreds of nodes, but thousands at a time.

BUILDING THE FOUNDATION OF SECURE CONTAINERS

In this discussion with Nathan McCauley of Docker, we talk about the different ways that users have come to think about Docker security over the years. McCauley explained that many early security concerns stemmed from users not being familiar with the technology at play. Docker has addressed these concerns in the base platform over the years, aided by a movement in the DevOps community towards embracing container technology and how it can help them achieve their security goals. The discussion also covers Docker 1.12's native Swarm orchestration mode, and some additional security features like cryptographic node identity.

Listen on SoundCloud or **Listen on YouTube**

Nathan McCauley *is director of security at Docker. His interests are cryptography, distributed trust and systems security. Before Docker he worked at Square and Arxan Technologies.*

NETWORKING, SECURITY & STORAGE DIRECTORY

NETWORKING

	Product/Project (Company or Supporting Org.)	Sub-category
	Avi Vantage Platform (Avi Networks)	Overlays and Virtual Networking Tools
	A software-based application delivery solution that integrates with container-based environments to provide dynamically configured load balancing, service discovery, application mapping, and auto-scaling capabilities. It also provides monitoring at the service-fabric level.	
	Aviatrix (Aviatrix)	
	Securely connects container groups and enforcing policies across distributed hybrid and public clouds.	
	Big Cloud Fabric (Big Switch Networks)	
	A data center fabric built using open networking switches and SDN controller technology. It enables container networking with DC/OS, Docker and Kubernetes.	
	Bluemix Virtual Private Network (VPN) (IBM)	
	Provides a secure IP-layer connectivity between your on-premises data center and your Bluemix cloud. An IPsec-compatible VPN gateway is required in your on-premises data center.	
Open Source	**Canal (Tigera)**	Overlays and Virtual Networking Tools
	A community-driven initiative that integrates the Calico and flannel networking projects. Applying Calico's industry-leading network policy enforcement capabilities to flannel's broad range of connectivity options, Canal represents the best-of-breed policy-based networking solution for cloud-native applications.	
Open Source	**Cisco Application Centric Infrastructure (Cisco)**	Overlays and Virtual Networking Tools
	ACI is used within Nexus switches. It uses a software-defined networking policy model across networks, servers, storage, security and services.	
Open Source	**Container Network Interface (CNI) (N/A)**	
	CNI is a project to help configure network interfaces for Linux application containers. It helps set up network connectivity of containers and remove allocated resources when the container is deleted.	
Open Source	**Contiv Network (Cisco)**	
	Contiv Network's Netplugin provides container networking for various use cases, including Kubernetes. It is part of Project Contiv, an open source project defining infrastructure operational policies for container-based application deployment.	
	Diamanti (Diamanti)	

	Product/Project (Company or Supporting Org.)	Sub-category
	A purpose-built container infrastructure that addresses the challenges of deploying containers to production while letting users keep their existing infrastructure. They do switching native on bare metal, plugging into a CPU bus.	
Open Source	**Flannel (CoreOS)**	Overlays and Virtual Networking Tools
	An etcd-backed network fabric for hosting containers. Flannel is a virtual network that gives a subnet to each host for use with container runtimes.	
	Joyent Triton (Joyent)	
	A containers-as-a-service offering, Triton comes with ContainerPilot, DataCenter and SmartOS functionality built-in.	
Open Source	**Joyent Triton DataCenter (Joyent)**	
	An open source cloud management platform, optimized to deliver next generation, container-based, service-oriented infrastructure across one or more data centers. Triton DataCenter converges container orchestration and cloud orchestration into a simple to deploy and manage solution. It provides user-defined networking, which makes it easy to create private layer 2 and layer 3 networks. Furthermore, it has integrated DNS and integrated per-instance firewalls.	
Open Source	**Joyent Triton SmartOS (Joyent)**	
	A lightweight container hypervisor that delivers industrial-grade security, networking, storage and performance to containers. Triton SmartOS has built-in storage and networking capabilities.	
Open Source	**Kuryr (OpenStack Foundation)**	Overlays and Virtual Networking Tools
	A Docker network plugin that uses OpenStack Neutron to provide networking services to Docker containers. It provides containerized images for the common Neutron plugins.	
Open Source	**libnetwork (Docker)**	Overlays and Virtual Networking Tools
	libnetwork provides a Docker-native Go implementation for connecting containers. It is based on the Container Network Model (CNM), as opposed to the Container Network Interface (CNI) model.	
	Nuage Networks VSP (Virtualized Services Platform) (Nokia)	Overlays and Virtual Networking Tools
	Provides software-defined networking capabilities for clouds of all sizes. It is implemented as a non-disruptive overlay for all existing virtualized and non-virtualized server and network resources. It provides support for Docker containers and works with OpenStack infrastructure.	
Open Source	**OpenContrail (Juniper Networks)**	
	An Apache 2.0-licensed project that is built using standards-based protocols and provides all the necessary components for network virtualization: SDN controller, virtual router, analytics engine, and published northbound APIs. It has an extensive REST API to configure and gather operational and analytics data from the system.	
	PLUMgrid Open Networking Suite (PLUMgrid)	Overlays and Virtual Networking Tools
	Helps enterprises and service providers operationalize their OpenStack cloud's virtual network and SDN deployments. PLUMgrid ONS supports Project Kuryr.	

	Product/Project (Company or Supporting Org.)	Sub-category
Open Source	**Project Calico (Tigera)**	Overlays and Virtual Networking Tools
	Provides a scalable networking solution for connecting data center workloads (containers, VMs or bare metal). It uses a Layer 3 approach. Calico can be deployed without encapsulation or overlays to provide high performance at massive scales.	
Open Source	**Project Skyhook (Aviatrix)**	
	It enables policy-based networking of container groups. Project Skyhook is part of Aviatrix's end-to-end encrypted cloud networking and management software solution.	
	Robin Systems (Robin Systems)	
	A container-based, application-aware compute and storage platform that abstracts the underlying operating platform from applications by making the servers, VMs, and storage boundaries invisible. Robin Systems is built using commodity hardware.	
Open Source	**Romana (N/A)**	Overlays and Virtual Networking Tools
	A network and security automation solution for cloud-native applications. Romana automates the creation of isolated cloud-native networks and secures applications with a distributed firewall that applies access control policies consistently across all endpoints and services, wherever they run.	
Open Source	**VPNKit (Docker)**	Overlays and Virtual Networking Tools
	A set of tools and services for helping HyperKit VMs interoperate with host VPN configurations.	
	Weave Net (Weaveworks)	Overlays and Virtual Networking Tools
	Connects containers into a transparent, dynamic and resilient mesh. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery.	

SECURITY

	Product/Project (Company or Supporting Org.)	Sub-category:
	Amazon EC2 Container Registry (ECR) (Amazon Web Services)	Registry-related
	A fully-managed container registry to store, manage and deploy Docker container images. It is included in Amazon EC2 Container Service (ECS) and integrates with AWS Identity and Access Management (IAM).	
Open Source	**Anchore (Anchore)**	Vulnerability Scanner or Image Analysis
	A set of tools to provide visibility, transparency and control to your container environment. It consists of two parts: a web service hosted by Anchore, and a set of open source command-line query tools. The hosted service selects and analyzes popular container images from Docker Hub and other sources, and provides this metadata as a service to the on-premises command-line tools.	
	Apcera Platform (Apcera)	Policy/Compliance Management
	Apcera manages access to compute resources across a cluster of servers. By focusing on managing policies across multiple environments, it aims to secure workloads and containers in enterprise production environments.	
	Aqua Container Security Platform (Aqua Security Software)	Vulnerability Scanner or Image Analysis
	Provides a scalable security solution that protects containerized applications against internal and external threats.	
	Aqua Peekr (Aqua Security Software)	Vulnerability Scanner or Image Analysis
	Free scanner of container images across different types of registries.	
Open Source	**Atomic Registry (Red Hat)**	Registry-related
	An open source enterprise registry based on the Origin and Cockpit projects, enhancing the Docker registry library.	
Open Source	**Atomic Scan (Red Hat)**	Vulnerability Scanner or Image Analysis
	The atomic scan function can be used instead of OpenSCAP, which is Project Atomic's default vulnerability scanner.	
	Aviatrix (Aviatrix)	Policy/Compliance Management
	Securely connects container groups and enforcing policies across distributed hybrid and public clouds.	

	Product/Project (Company or Supporting Org.)	Sub-category:
	BanyanOps (BanyanOps)	Vulnerability Scanner or Image Analysis
	The company has yet to launch its product, which will focus on analyzing images. BanyanOps wants to accelerate IT operations with containers.	
	Bluemix (IBM)	Vulnerability Scanner or Image Analysis
	Bluemix is IBM's digital application platform, providing all you need to develop, deploy, manage and run cloud applications. Bluemix storage services are built on OpenStack Swift and allow for management of objects and block volumes through the OpenStack API. The Bluemix VPN and Secure Gateway services are just two offerings that enable container networking. Select your prefered managed platform for your applications from Containers, Virtual Machines and Cloud Foundry for instant run times. Freedom to develop microservices-based applications within a platform or with microservices deployed across multiple platforms, using Bluemix services for deployment, operations, logging and monitoring.	
Open Source	**Canal (Tigera)**	Policy/Compliance Management
	A community-driven initiative that integrates the Calico and flannel networking projects. Applying Calico's industry-leading network policy enforcement capabilities to flannel's broad range of connectivity options, Canal represents the best-of-breed policy-based networking solution for cloud-native applications.	
Open Source	**Cisco Application Centric Infrastructure (Cisco)**	Policy/Compliance Management
	ACI is used within Nexus switches. It uses a software-defined networking policy model across networks, servers, storage, security and services.	
Open Source	**Clair (CoreOS)**	Vulnerability Scanner or Image Analysis
	A container vulnerability analysis service providing static analysis of vulnerabilities in appc and Docker containers.	
	CloudPassage Halo (CloudPassage)	Vulnerability Scanner or Image Analysis
	Provides instant visibility and continuous protection for servers in any combination of data centers, private clouds and public clouds.	
	Conjur (Conjur)	
	A single source of secrets management and orchestration so you can easily store, fetch, update, and distribute secrets to your entire infrastructure. In addition, Conjur's Dynamic Traffic Authorization is built on the ability to assign identity and apply role-based policies to machines and processes. Once enrolled and assigned an identity, machines can be managed as part of a group, or a layer of hosts, including containers.	
Open Source	**Docker Bench for Security (Docker)**	
	The Docker Bench for Security is a script that checks for dozens of common best practices around deploying Docker containers in production.	
	Docker Cloud (Docker)	Vulnerability Scanner or Image Analysis
	A SaaS service for deploying and managing Dockerized applications. Docker Cloud includes Docker Security Scanning, which reviews images in private repositories to verify that they are free from known security vulnerabilities or exposures, and report the results of the scan for each image tag. Docker Trusted Registry is included in the subscription.	

	Product/Project (Company or Supporting Org.)	Sub-category:
	Docker Datacenter (Docker)	Registry-related
	Provides on-premises container management and deployment services to enterprises with a production-ready platform supported by Docker and hosted locally behind the firewall. Docker Trusted Registry is included in the subscription.	
Open Source	**Docker Distribution (Docker)**	Registry-related
	A toolset to pack, ship, store and deliver content. It supersedes the docker/docker-registry project. Its main component is the Docker Registry 2.0 implementation for storing and distributing Docker images.	
Open Source	**Docker Engine (Docker)**	
	A lightweight runtime and tooling that builds and runs your Docker containers. CS Docker Engine is the commercially supported version. Docker Content Trust is a feature that makes it possible to verify the publisher of Docker images.	
	Docker Store (Docker)	Vulnerability Scanner or Image Analysis
	A place to find trusted commercial and free software distributed as Docker images. All Docker official images and Docker Store-curated content goes through Docker Security Scanning.	
	Docker Trusted Registry (Docker)	Registry-related
	Allows users to store and manage Docker images on premises or in a virtual private cloud.	
Open Source	**Dockyard (N/A)**	Registry-related
	An image hub for Docker, rkt or other container engines.	
	Enterprise Registry (CoreOS)	Registry-related
	Provides a secure registry on an enterprise's own infrastructure.	
	FlawCheck Private Registry (FlawCheck)	Vulnerability Scanner or Image Analysis
	A cloud-hosted container registry that hosts Docker containers. It scans containers for vulnerabilities and malware.	
	FlawCheck Private Registry Enterprise (FlawCheck)	Vulnerability Scanner or Image Analysis
	An on-premise container registry that hosts Docker containers. It scans containers for vulnerabilities and malware.	
	Google Container Registry (Google)	Registry-related
	Provides secure, private Docker image storage on Google Cloud Platform.	
Open Source	**Harbor (VMware)**	Registry-related
	An enterprise-class registry server. It extends the open source Docker Registry server by adding more functionalities usually required by an enterprise. Harbor is designed to be deployed in the private environment of an organization.	
	Illumio Adaptive Security Platform (ASP) (Illumio)	

	Product/Project (Company or Supporting Org.)	Sub-category:
	ASP is a distributed software platform designed to continuously protect communications within and across tiers of applications, wherever they are running. It creates secure and granular segmentation to compartmentalize workloads and applications, reducing the attack surface exposed to cyber vulnerabilities. It supports all modern server computing formats (Windows/Linux, virtual machines, containers) and all computing environments (data center, private and public cloud).	
Open Source	**Joyent Triton SmartOS (Joyent)** A lightweight container hypervisor that delivers industrial-grade security, networking, storage and performance to containers. Triton SmartOS has built-in storage and networking capabilities.	
Open Source	**Joyent Triton DataCenter (Joyent)** An open source cloud management platform, optimized to deliver next generation, container-based, service-oriented infrastructure across one or more data centers. Triton DataCenter converges container orchestration and cloud orchestration into a simple to deploy and manage solution. It provides user-defined networking, which makes it easy to create private layer 2 and layer 3 networks. Furthermore, it has integrated DNS and integrated per-instance firewalls.	
Open Source	**Notary (Docker)** The Notary project is comprised of a server and a client for running and interacting with trusted collections. Publishers can sign their content offline using highly secure keys.	Registry-related
Open Source	**OpenSCAP (Red Hat)** The OpenSCAP Base is both a library and a command line tool which can be used to parse and evaluate each component of the SCAP (Security Content Automation Protocol) standard for describing system configurations and security management policies. OpenSCAP also includes a GUI and tool allowing users to perform configuration and vulnerability scans on a single system.	Vulnerability Scanner or Image Analysis
	Polyverse (Polyverse) Uses millions of individually protected containers to help prevent large-scale data breaches.	
Open Source	**Portus (SUSE)** Acts both as an authorization server and user interface for Docker registry V2.	Registry-related
	Private Image Registry Service (IBM) IBM Containers on Bluemix provide a private Docker image registry service for hosting private images. The private registry supports group access policies to allow teams to share private images.	Registry-related
Open Source	**Project Calico (Tigera)** Provides a scalable networking solution for connecting data center workloads (containers, VMs or bare metal). It uses a Layer 3 approach. Calico can be deployed without encapsulation or overlays to provide high performance at massive scales.	Policy/Compliance Management
Open Source	**Project Skyhook (Aviatrix)** It enables policy-based networking of container groups. Project Skyhook is part of Aviatrix's end-to-end encrypted cloud networking and management software solution.	Policy/Compliance Management

	Product/Project (Company or Supporting Org.)	Sub-category:
	Quay Enterprise (CoreOS)	Registry-related
	Provides cloud-based, secure hosting for private Docker repositories.	
	Quay.io (CoreOS)	Registry-related
	A secure image registry run on your own servers.	
Open Source	**Registrator (Glider Labs)**	Registry-related
	Registrator automatically registers and deregisters services for any Docker container. It supports pluggable service registries like Consul and etcd.	
	Twistlock Runtime (Twistlock)	Policy/Compliance Management
	Enforces runtime policies, performs access control, detects compromises and protects your containerized application from active threats.	
	Twistlock Trust (Twistlock)	Vulnerability Scanner or Image Analysis
	Scans images and registries to detect vulnerabilities in the code as well as configuration errors. It gives security teams a centralized location to configure and monitor security rules across multiple container clusters.	
	Vulnerability Advisor (IBM)	Vulnerability Scanner or Image Analysis
	Vulnerability Advisor scans each layer in every Docker image in the user's private hosted registry for the latest known vulnerabilities and policy settings, providing insight to the security posture before instantiating a live container. The administrator can set deployment policy based on an image's security compliance.	

STORAGE

	Product/Project (Company or Supporting Org.)	Sub-category:
	Acropolis (Nutanix)	Hardware or Cloud Offering
	Scale-out data fabric for storage, compute and virtualization. Acropolis Container Services (ACS) provides the ability to easily deploy and manage containerized applications on the Nutanix platform. It includes two core capabilities, persistent container volumes and the Nutanix Docker machine driver. Acropolis can be purchased as hardware or licensed as software.	
	Bluemix (IBM)	
	Bluemix is IBM's digital application platform, providing all you need to develop, deploy, manage and run cloud applications. Bluemix storage services are built on OpenStack Swift and allow for management of objects and block volumes through the OpenStack API. The Bluemix VPN and Secure Gateway services are just two offerings that enable container networking. Select your prefered managed platform for your applications from Containers, Virtual Machines and Cloud Foundry for instant run times. Freedom to develop microservices-based applications within a platform or with microservices deployed across multiple platforms, using Bluemix services for deployment, operations, logging and monitoring.	
	Bluemix Object Storage (IBM)	Hardware or Cloud Offering
	The IBM Bluemix Object Storage service provides access to a fully provisioned Swift Object Storage account to manage your data. Swift provides a fully distributed, API-accessible storage platform.	
Open Source	**Ceph (Red Hat)**	
	A distributed object store and file system designed to provide excellent performance, reliability and scalability.	
	ClusterHQ Volume Hub (ClusterHQ)	Management or Data Volume
	A free tool for creating, viewing and monitoring Flocker clusters.	
Open Source	**Contiv Storage (Cisco)**	
	Via Ceph, policy-backed clustered storage for Docker. It is part of Project Contiv, an open source project defining infrastructure operational policies for container-based application deployment.	
Open Source	**Convoy (Rancher Labs)**	Management or Data Volume
	A Docker volume plugin, managing persistent container volumes.	
Open Source	**Crate (Crate.io)**	
	Uses SQL syntax for distributed queries across a cluster. Crate is masterless and its "shared nothing architecture" means that all nodes are equal.	

	Product/Project (Company or Supporting Org.)	Sub-category:
	Datera Elastic Data Fabric (Datera)	Hardware or Cloud Offering
	Scale-out storage software that turns standard, commodity hardware into a RESTful API-driven, policy-based storage fabric for large-scale clouds.	
	Diamanti (Diamanti)	Hardware or Cloud Offering
	A purpose-built container infrastructure that addresses the challenges of deploying containers to production while letting users keep their existing infrastructure. They do switching native on bare metal, plugging into a CPU bus.	
Open Source	**dvol (ClusterHQ)**	Management or Data Volume
	Version control for development databases within Docker.	
Open Source	**Flocker (ClusterHQ)**	Management or Data Volume
	A data volume manager for Dockerized applications.	
	Hedvig Distributed Storage Platform (Hedvig)	Hardware or Cloud Offering
	A single unified solution that lets you tailor a modern, high-performance, elastic storage system built with low-cost commodity hardware to support any application, hypervisor, container or cloud.	
	IBM Cloud Object Storage (IBM)	Hardware or Cloud Offering
	IBM bought Cleversafe in late 2015. It can be deployed as a Docker container.	
	Joyent Triton (Joyent)	Hardware or Cloud Offering
	A containers-as-a-service offering, Triton comes with ContainerPilot, DataCenter and SmartOS functionality built-in.	
Open Source	**Joyent Triton SmartOS (Joyent)**	
	A lightweight container hypervisor that delivers industrial-grade security, networking, storage and performance to containers. Triton SmartOS has built-in storage and networking capabilities.	
Open Source	**Kubernetes (Cloud Native Computing Foundation)**	
	Kubernetes is an open source Docker orchestration tool. Google initially developed Kubernetes to help manage its own LXC containers. Stateful support is done through a new object called Pet Set. In addition, there are many networking and data volume plug-ins available.	
Open Source	**libstorage (EMC)**	
	A portable and remote storage plugin framework. It provides a vendor agnostic storage orchestration model, API, and reference client and server implementations.	
Open Source	**Manta (Joyent)**	Hardware or Cloud Offering
	Joyent offers Manta via a cloud service and with a licensed and supported enterprise version. Manta is an HTTP-based object store that uses OS containers to allow running arbitrary compute on data at rest. Manta Functions enable serverless computing: executing code, on demand, on Manta Objects without having to move your data into a separate computing environment.	

	Product/Project (Company or Supporting Org.)	Sub-category:
Open Source	**Pachyderm (Pachyderm)** Enables storage and analysis of large data sets using containers. Pachyderm consists of a file system and pipeline. The file system allows version control of data sets and the pipeline is a processing engine for the versioned data.	
Open Source	**Polly (EMC)** Volume scheduling for container schedulers.	Management or Data Volume
Open Source	**Portworx PX-Developer (Portworx)** Deploying the PX-Developer container on a server with Docker Engine turns that server into a scale-out storage node. Storage runs converged with compute and gives bare metal drive performance.	
	Portworx PX-Enterprise (Portworx) PX-Enterprise is scale-out storage that deploys as a container. As opposed to Portworx's developer offering, it is sold as an appliance.	Hardware or Cloud Offering
	Quobyte (Quobyte) A fault-tolerant and scalable file system. It is software-defined storage software that turns standard server hardware into a data center file system. It draws on work done for the open source distributed file system XtreemFS.	
Open Source	**REX-Ray (EMC)** An abstraction layer between storage endpoints and container platforms. The administration and orchestration of various storage platforms can all be performed using the same set of commands.	
	Robin Systems (Robin Systems) A container-based, application-aware compute and storage platform that abstracts the underlying operating platform from applications by making the servers, VMs, and storage boundaries invisible. Robin Systems is built using commodity hardware.	Hardware or Cloud Offering
	StorageOS (StorageOS) Stateful, durable, highly available, software only, enterprise-ready persistent container storage. StorageOS provides enterprise storage array functionality delivered via software on a pay-as-you-go basis	
Open Source	**Torus (CoreOS)** Distributed storage coordinated through etcd. It ships with a simple block-device volume plugin, but is extensible to more.	Management or Data Volume

DISCLOSURES

The following companies mentioned in this ebook are sponsors of The New Stack: Apcera, Arcadia Data, Bitnami, Capital One, CloudFabrix, Cloud Foundry, Cloudsoft, Cloud Native Computing Foundation, Codenvy, CoreOS, DigitalOcean, Hewlett Packard Enterprise, Intel, Iron.io, Mesosphere, New Relic, Red Hat, Sysdig, Weaveworks.