Exploring a modern architecture using docker, kubernetes and microservices in Cisco eCommerce B2B

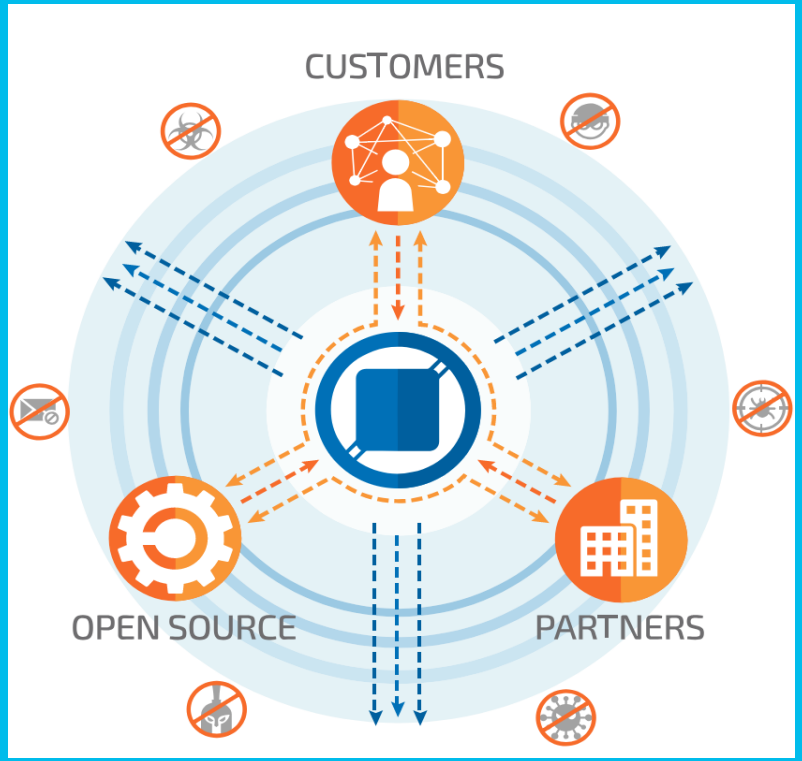
Jay Chow

Software Engineer, Cisco eCommerce, IT

Cisco, San Jose

23rd January 2020

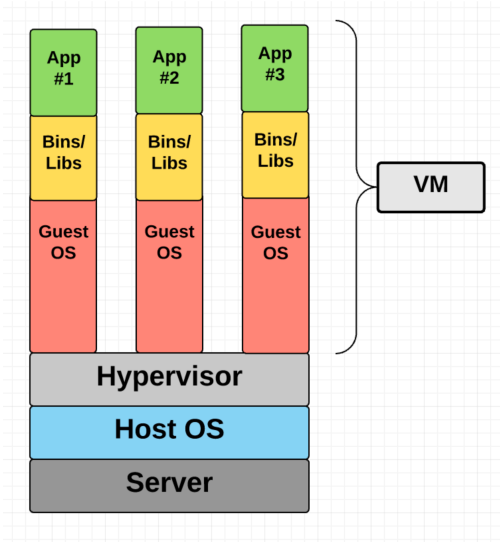
Agenda

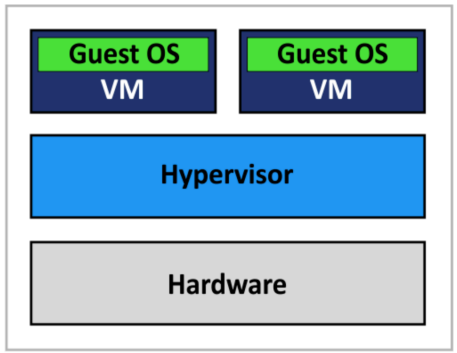


1. What is a virtual machine?
2. What is a container?
3. Docker
4. Kubernetes
5. Microservices
6. Kubernetes with microservices architecture
7. What's next for us?
8. Q & A
9. References

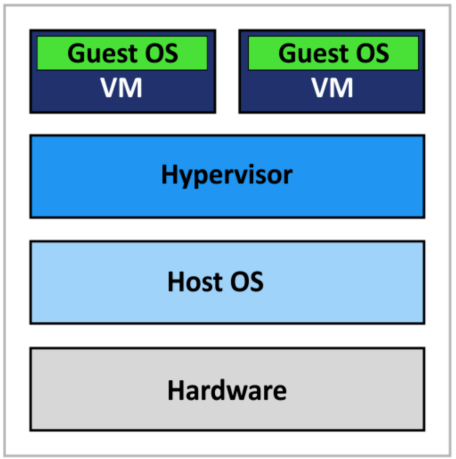
What is a virtual machine?

- Emulation of a real computer that executes programs
- Hardware level virtualization
- VMs run on top of a physical machine using a hypervisor(i.e VMware Fusion, Oracle Virtual Box)





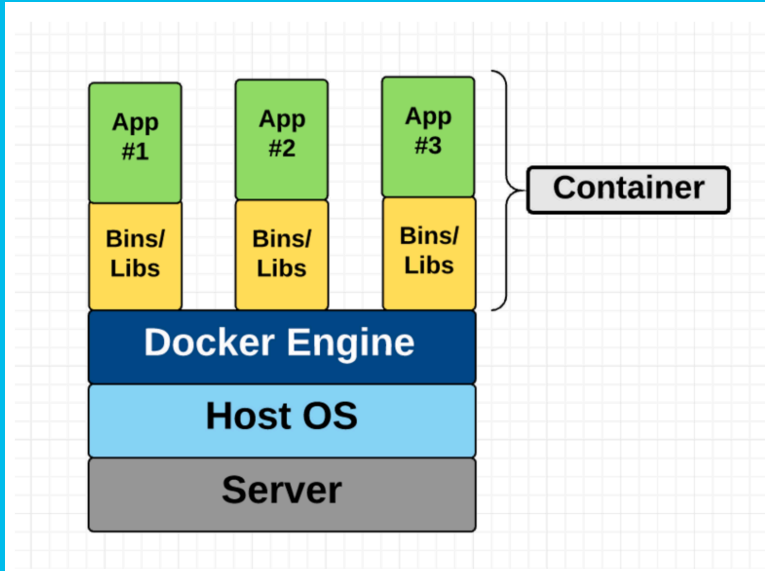
Type 1 Hypervisor
(Bare-Metal Architecture)



Type 2 Hypervisor
(Hosted Architecture)

What is a container?

- A container is just a method of packaging, deploying and running a linux process
- Container is a useful resource allocation and sharing technology
- Operating system level virtualization
- Each container gets isolated user space
- Multiple containers on single host machine



Container Diagram

What is a container?(cont'd)

- A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

What is an image?

- An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.
- You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Docker and why use it?

- Open-source project based on linux containers
- Uses linux kernel features like namespaces and control groups to create containers
- Ease of use: allows anyone to package an application on the laptop, which can be run on any public cloud, private cloud or bare metal – “build once, run anywhere”
- Speed: containers are lightweight and fast; containers are sandboxed environments running on a shared host OS, they consumes fewer resources(compared to VMs)
- Docker Hub: “app store for docker images”, contains many public images that are readily available, search for images, pull and then use with little-to-no modification
- Modularity and scalability: run a database service in container, run a web server in another container, making it easy to scale or update components independently

Components of Docker

Docker engine: lightweight runtime layer that manages the containers, images etc.
- compromises of docker daemon, docker client and a rest API for interacting with docker daemon remotely

1) Docker client: “UI” for docker, sends instructions to docker daemon

2) Docker daemon: executes commands from docker client

3) Dockerfile: write instructions to build a Docker image

4) Docker image: read-only template built from instructions in a Dockerfile

https://cloudpractice.cisco.com/tutorial/docker/

Note:
Docker containers are built off docker images

Kubernetes

- Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.
- With Kubernetes, able to quickly and efficiently respond to customer demand:
  - Deploy your applications quickly and predictably.
  - Scale your applications on the fly.
  - Seamlessly roll out new features.
  - Optimize use of your hardware by using only the resources you need.

Pod

A *pod* is a group of one or more containers, the shared storage for those containers, and options about how to run the containers.

Pods are always co-located and co-scheduled and run in a shared context. A pod models an application-specific “logical host” - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, they would have executed on the same physical or virtual machine.

Containers within a pod share an IP address and port space and can find each other via localhost. They can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different pods have distinct IP addresses and can not communicate by IPC.

Applications within a pod also have access to shared volumes, which are defined as part of a pod and are made available to be mounted into each application’s filesystem.

Node

- Node is a worker machine in Kubernetes, previously known as Minion.
- Node may be a VM or physical machine, depending on the cluster.
- Each node has the services necessary to run Pods and is managed by the master components
- The services on a node include docker, kubelet and network proxy.
  - Kublet is the primary node agent that runs on each node

Tradeoff for Kubernetes

Benefits:

- Increases infrastructure utilization through the efficient sharing of computing resources across multiple processes. Kubernetes is the master of *dynamically allocating computing resources to fill the demand*.

- This allows organizations to avoid paying for computing resources they are not using. However, there are side benefits that make the transition to microservices much easier.

Costs:

- Complex technology to learn and harder to manage

- https://wiki.cisco.com/display/IWAN3/Kubernetes

Microservices – software pattern design

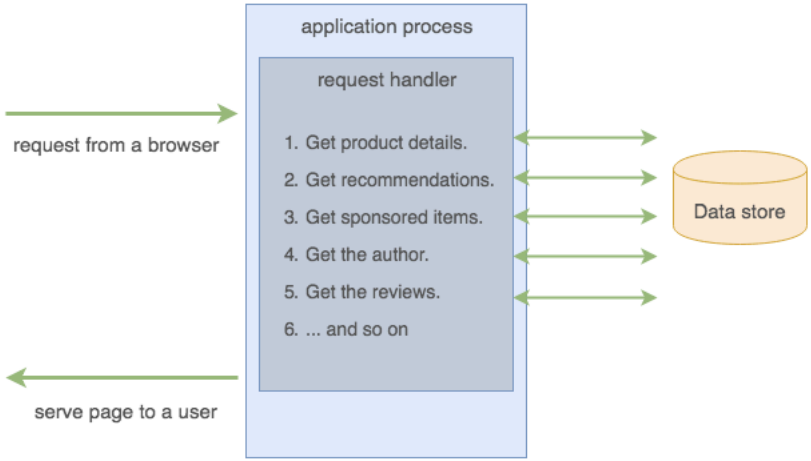
What is a microservice?

- Just a computer program which runs on a server or virtual computing instance and responds to network requests

- Narrower scope and focuses on doing smaller tasks

Let’s look at a use case from Amazon(next slide): Amazon Product Listing, the system that serves the product page on Amazon

Amazon Product Listing



When user request comes from browser, and the logic inside the handler will sequentially make several calls to different databases, fetch the required info needed to a render the web page to be returned to the user.

But what if the application grows and more engineers join different teams? Each team wants to:

1) Change their database schema.

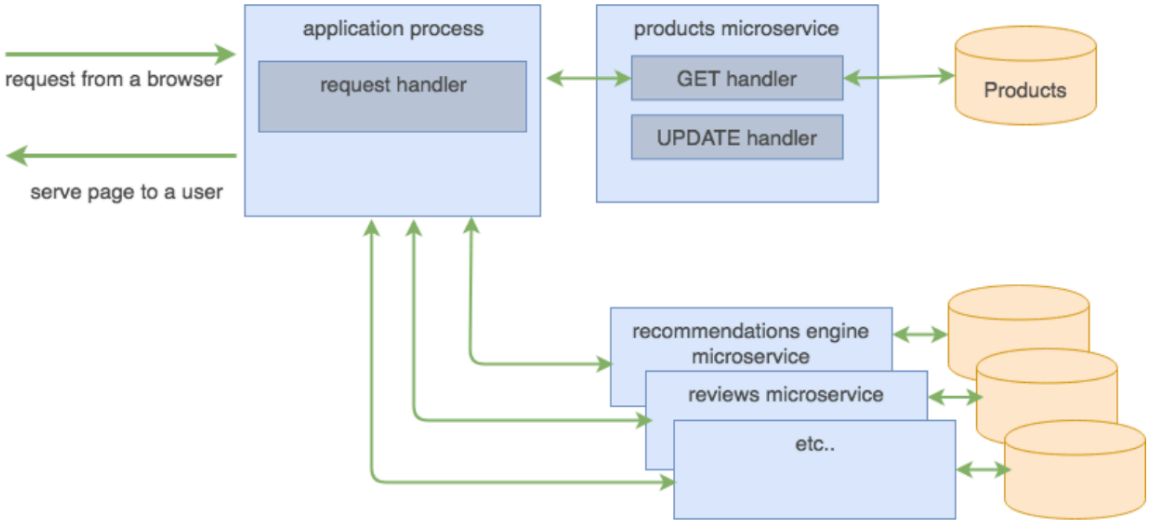
2) Release their code to production quickly and often.

3) Use development tools like programming languages or data stores of their choice.

4) Make their own trade-offs between computing resources and developer productivity.

5) Have a preference for maintaining/monitoring its own functionality.

Solution? Microservices!



Microservices are now separated out and each team working on its microservice can:

1) Deploy their service as frequently as they wish without disrupting other teams.
2) Scale their service the way they see fit(i.e use AWS instance types of their choice or perhaps run on specialized hardware).
3) Have their own monitoring, backups and disaster recovery that are specific to their service.

Tradeoff for Microservices

Benefits:

- Easier automated testing
- Rapid and flexible deployment models
- Higher overall resiliency

Costs:

- Need for more careful planning
- Higher R&D investment up front
- Over-engineering

Kubernetes with Microservices architecture

Solve problems like:

1) Predicting how much computing resources each service will need; how these requirements change under load

2) How to carve out infrastructure partitions and divide them between microservices; and enforce resource restrictions

3) Kubernetes solves these problems quite elegantly and provides a common framework to describe, inspect and reason about infrastructure resource sharing and utilization. Hence, adopting Kubernetes as part of your microservice re-architecture is a good idea

4) Kubernetes is a great platform for complex applications comprised of multiple microservices

Consider the open source solution, Gravity (https://github.com/gravitational/gravity), an open source Kubernetes packaging solution, which removes the need for Kubernetes administration

What’s next for us?

Let’s discuss!

Questions?

References

https://engineering.shopify.com/blogs/engineering/shopify-manages-api-versioning-breaking-changes
https://gravitational.com/blog/microservices-containers-kubernetes/
https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/
https://docs.docker.com/engine/reference/builder/
https://cloudpractice.cisco.com/tutorial/
https://engit.cisco.com/storage-and-compute/cisco-crate
https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/