Plagiarism Scan Report

Content Checked For Plagiarism:

The Era of Containerization

You must have come across virtual machines and probably used one for running a Linux machine inside windows. The good old days of virtual box and VMware when running a virtual machine needed considerable hardware resources, and once you switch to the Guest OS, you could barely do anything on the Host OS. Multiple VMs running together was a far stretched dream.

In the early days, virtual machines served the purpose of stable development environment over a team. A virtual machine with all the required software was made and passed on to each team member.
A multi gigabyte Operating system image was first installed, then all other needed softwares were configured, let's say, the standard web development package tools were downloaded and installed, a few libraries integrated, the FTP server configured, database setup. All this was done for one system. Then copy over the virtual machine file, and hope that it works well on other systems as well. Pretty easy, right?
This set up was an overhead. Even for a standard web development environment, the devs had to go through all this. We needed something better.

Enter vagrant.
Vagrant brought the concept of boxes, which were VM images with standard dev environments pre-installed. Setting up the new tech stack was now a command away. Just find the right box, and start working. Vagrant leverages a declarative configuration file which describes all your software requirements. Instead of passing around VM's between team members, this config file could be version controlled, everyone could just pull and get ready to work. Starting a configured VM was as easy as typing `vagrant up`.

Vagrant, for long, gave us consistent environments to develop, test and deploy our app. But VM's were inherently slow, firing up the smallest of them would take a few minutes.
Just because I wanted to try a software with a clean slate, I had to install an OS, along with all of its features, the file system, the drivers. A lot of this goes utilized. We wanted something lighter, which would just do just what we wanted, and nothing more.

Enter Docker.
Docker Inc is the company which popularised the concept of containers. Think of a

container to be same as a container from a cargo ship. we don't know what's inside of each box, and from outside they all look same. what's inside them is limited to them. They are significantly lighter than VMs.

A virtual machine packs together a guest OS, its kernel, file system, device drives, then our tools, runtimes, and more. This makes them heavy.

Docker used the fact that we don't really want an OS. All we want is a dev environment. It leverages this fact, makes use of the Host OS kernel, drivers and more instead of packing everything fresh in Guest OS. This results in a smaller size and much faster deployment of containers.

Like vagrant boxes, Docker also utilizes the concept of centrally hosted images. Docker Hub is the play store equivalent for dev environments. You find `images` which fit your need, pull them, and then spin out containers from them. Containers are like runtimes of the images. so anything you do in the container gets lost as soon as you take them down. But that's not what we wanted, right? For this, Docker gives us Volumes, which are just shared folders between host OS and container. This gives us data persistence over container runs. Any changes made from within the container are saved to this volume.

Docker utilizes AuFS.

AuFS is a kind of union file system. The read-only part, which is common for all the containers is merged together with the write part, which is separate for each container. Thus, we only need write space if we want to add additional containers.

Requirement: 1000 VMs
Space: Space needed for 1 VM (1 GB) * 1000
Total: That's about 1000 GB.

Requirement: 1000 Containers
Space: (read-only space of the 1 GB OS) * 1 + (write-only space) * 1000
Total: This amounts to a little over 1GB and significantly less than the former.
---

one process per container Approach

With Docker, a rather new approach came into the picture, `one process per container`. Each container with a specific process within. so while building our custom app, we could just pick the images we want from docker hub.

Each image serving only one need, may it be database, or server, and so on.

This `one process per container` approach gives us horizontal scalability. Too many server requests? Need one more apache server? Just spin up one more container from the same image. It keeps the containers light, and in a state of failure, another one with same specs could be brought up within seconds.

This unveils the real power of Docker. Hundreds of containers, serving from the same image.

But, what if your application keeps growing? Let's say you keep adding features to the same codebase until it becomes a so big that is almost impossible to maintain and eats way too much RAM and CPU resources.

What we do is split it into smaller chunks, each responsible for one specific task,

maintained by a team, aka. microservices.
We can set up multiple instances of these microservices using containers to make it readily available in the production environment. We now have to deal with Load Balancing, storage management, Health checks, Auto-[scaling/restart/healing] of containers and nodes. Orchestration tools make this possible.

Container Orchestration refers to the automated arrangement, coordination, and management of software containers. Kubernetes, AWS ECS, and Docker Swarm are some of the leaders in Orchestration. Kubernetes has the largest community and is the most popular because it is built on Google's experience of a busy production environment. The availability of such tools makes the ideology behind containers much more powerful and practical for real-life applications. Self Healing of failed nodes, load balancing, scaling, updates, rollbacks, you name it.

Report generated by smallseotools.com