Plagiarism Scan Report

Summary	
Report Genrated Date	14 Mar, 2018
Plagiarism Status	**78% Unique**
Total Words	946
Total Characters	5675
Any Ignore Url Used	

Content Checked For Plagiarism:

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. Download the multi-gigabyte OS image that you want, install it, then download and configure the stack you'll be working with: let's say Apache, MySQL, PHP, then install some libraries, install an FTP server. Once this is all done, copy the code over, import the database, configure Apache's virtual host, restart and cross your fingers. Pretty easy, right?
This set up was an overhead. Even for a standard web development environment, the devs had to go through all this pain. 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 o 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, □ile system, device drives, then our tools, runtimes, and more. This makes them heavy.
Docker used the □act that we don't really want an OS. All we want is a dev environment. It leverages this □act, makes use o□ the Host OS kernel, drivers and more instead o□ packing everything □resh in Guest OS. This results in a smaller size and much □aster deployment o containers.

Like vagrant boxes, Docker also utilizes the concept o□ centrally hosted images. Docker Hub is the play store equivalent □or dev environments. You □ind `images` which □it your need, pull them, and then spin out containers □rom them. Containers are like runtimes o 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 □olders between host OS and container. This gives us data persistence over container runs. Any changes made □rom within the container are saved to this volume.

Docker utilizes AuFS.
AuFS is a layered □ile system, so you can have a read-only part and a write part which are merged together. The common parts o□ the operating system can be marked as read-only and shared amongst all o□ your containers and each container can be given its own mount □or writing.

one process per container Approach

With Docker, a rather new approach came into the picture, `one process per container`. Each container with a speci□ic process within. so while building our custom app, we could just pick the images we want □rom 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 □rom the same image. It keeps the containers light, and in a state o□ □ailure, another one with same specs could be brought up within seconds.
This unveils the real power o□ Docker. Hundreds o□ containers, serving □rom the same image.

But, what i□ your application keeps growing? Let’s say you keep adding more and more □unctionality until it becomes a massive monolith that is almost impossible to maintain and eats way too much CPU and RAM.
What we do is split it into smaller chunks, each responsible □or one speci□ic task, maintained by a team, aka. microservices.
We can run multiple instances o□ each microservice spanning across multiple servers to make it highly available in a production environment. We now have to deal with Load Balancing, storage management, Health checks, Auto-[scaling/restart/healing] o containers and nodes. Orchestration tools make this possible.

Container Orchestration

Container Orchestration re□ers to the automated arrangement, coordination, and management o□ so□tware containers. Kubernetes, AWS ECS, and Docker Swarm are some o□ the leaders in Orchestration. Kubernetes has the largest community and is the most popular.It is based on Google's experience o□ running workloads at a huge scale in

production over the past 15 years.
The availability o□ such tools makes the ideology behind containers much more power□ul and practical □or real-li□e applications. Sel□ Healing o□ □ailed nodes, load balancing, scaling, updates, rollbacks, you name.