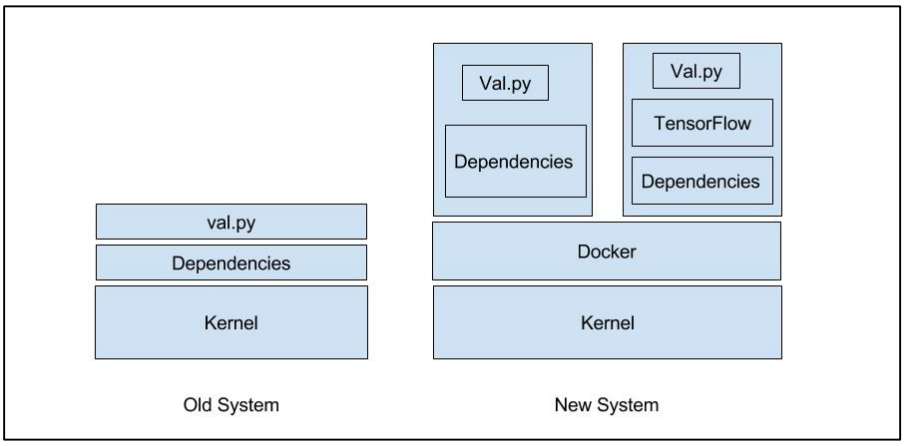
CONTAINERS

1. Why software containers?

In an old system, kernel in a machine is a bridge between applications and the actual data processing done at the hardware level. Application dependencies are the libraries other than your project code that are required to create and run your application. In a way, the laptop that we use for creating model are following this same basic architecture. Python engine and its dependencies are installed on our machines. Python codes run using the application which are managed by kernel on our machines. The improvement over this system was the evolution of Virtual Machines (VM). System virtual machines provide a substitute for a real machine. They provide functionality needed to execute entire operating systems. It consists of a hypervisor which uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another to exist on the same physical machine. The drawback of this system was memory allocation could not be optimized according to utilization, as each user was allocated a memory space that could or could not be utilized. An Amazon EC2 instance is like a VM on cloud. For this project, each user will have the capability to compute using TensorFlow on cloud, which will require multiple VMs, resulting in waste.

Further improvement made to solve this problem, led to the creation of Containers. Each container- uses same memory, disk space – same kernel. Thus, utilization is high. Each container has the App and its dependencies. Containers promise a streamlined, easy-to-deploy and secure method of implementing specific infrastructure requirements, and they also offer an alternative to virtual machines. Unlike virtual machines, they don't need a full OS to be installed within the container, and they don't need a virtual copy of the host server's hardware. Containers are able to operate with the minimum amount of resources to perform the task they were designed for; this can mean just a few pieces of software, libraries and the basics of an OS. This results in two or three times as many containers being able to be deployed on a server than virtual machines.

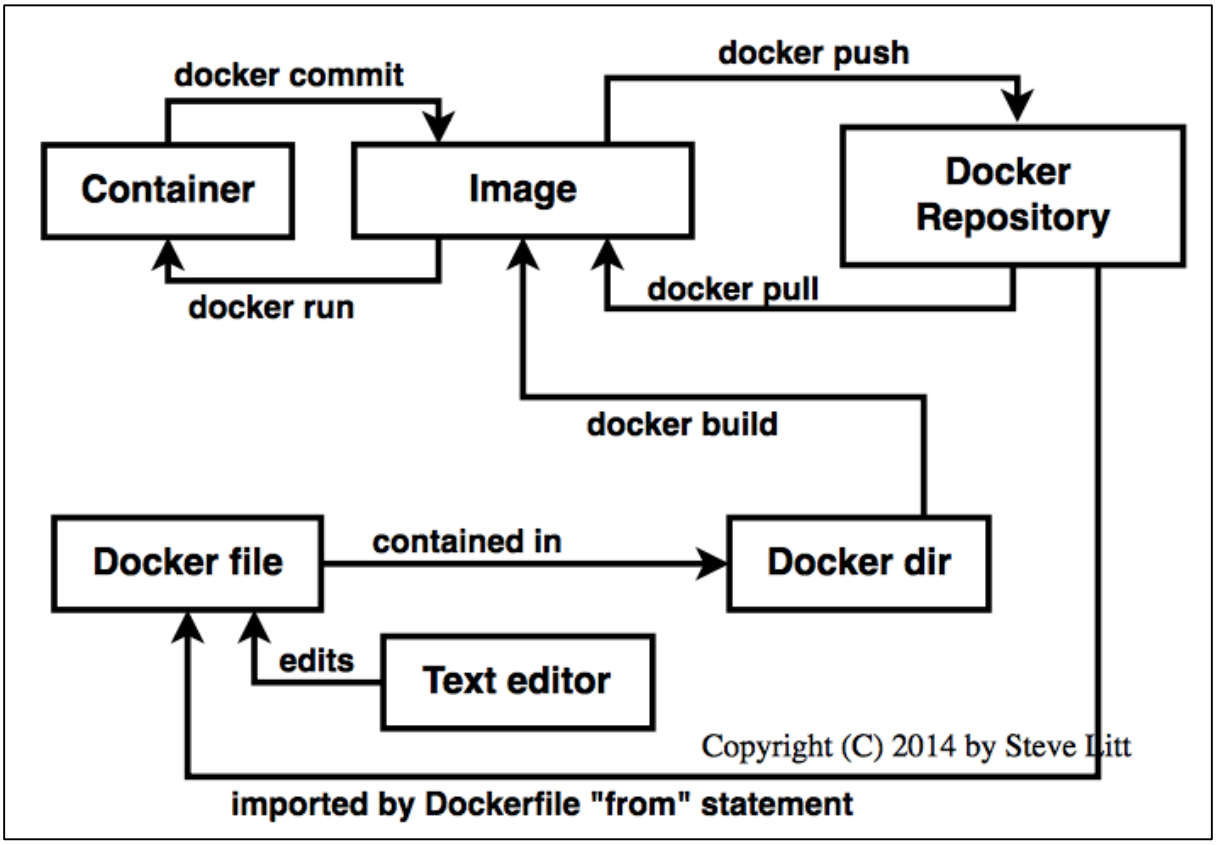


The diagram shows the old system and a comparable new system of Containers

Docker containers allow us to perform all the computations required to be done on a large scale using the same code as that on machine on which it was coded. This is important for our architecture design as Docker containers are also very portable, it can be deployed to different servers very easily. From a software lifecycle perspective this is great, as containers can be copied to create development, test, integration and live environments very quickly. From a software- and security-testing perspective this has a large advantage, because it ensures that the underlying OS is not causing a difference in the test results. This especially becomes important as we follow a Continuous Integration and Continuous Deployment approach to software development. Continuous Integration is a development process where team members are integrating their work frequently, often multiple times a day on a repository based system.

2. Introduction to Docker

A Docker system can be explained quite simply from the following image:



Some of the jargons associated with Docker are as follows:

a) Dockerfile

A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.

b) Container

A container is a runtime instance of a Docker image. A Docker container consists of

- A Docker image
- An execution environment
- A standard set of instructions

c) Image

Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.

d) Repository

A repository is a set of Docker images. A repository can be shared by pushing it to a registry server. The different images in the repository can be labeled using tags.

e) Registry

A Registry is a hosted service containing repositories of images which responds to the Registry API.

f) Tag

A tag is a label applied to a Docker image in a repository. Tags are how various images in a repository are distinguished from each other.

3. "Docker-izing" a pipeline stage

The process for working with Docker system is as follows:

1. Develop application – We will first write the application to detect objects from security camera videos. We will do this using TensorFlow in Python. The application should also classify image as a valid or invalid one before it is processed.

2. Build a Docker image for the app with Docker engine – We will then build the Docker image for the application that we developed based on Python engine.

3. Upload the image to a registry – The image is to be pushed to a registry server, where each image will be labeled using tags. We can have different images created for capturing threats depending on the client's requirement. Separate instances can work on 'images' from different clients in parallel.

4. Deploy a Docker container, based on the image and from the registry, to a VM – The image when run according to some set of instructions will detect the objects in the 'image' file that we provide at the input. Thus, the application will create an output file with objects, probability of that being the identified object, time when it was detected and the location where the 'image' came from.

Docker Hub is public registry for Docker Image, so we can get the base image to use for python in Docker. There is official python image that can be found on 'www.hub.docker.com'. We would need to add the application code that we wrote on TensorFlow to the image. There is no need to start from scratch. Smaller the size of images (<5Mb), better is the portability. The Python base image is around 200 Mb and is considered as a medium size image.

4. Hand on! Working with Docker

On initializing Docker by "`sudo docker images`" we get the list of repositories, the tags for the images, image id, time when the image was created and the size of image.

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              20c44cd7596f        3 days ago          123MB
ubuntu              xenial              20c44cd7596f        3 days ago          123MB
ubuntu              trusty              d6ed29ffda6b        3 days ago          221MB
alpine              latest              053cde6e8953        2 weeks ago         3.97M
```

We all were connected to the same instance on cloud. Docker allows us to run script on machine as Docker container. Thus, instead of installing all the files required to run that file, we can execute that file directly on cloud. The code used to download the image built by dwhitena from mgmt-validate is "`sudo docker pull dwhitena/mgmt-validate:slim`"

```
slim: Pulling from dwhitena/mgmt-validate
b56ae66c2937: Already exists
ebd41a1f8f00: Pull complete
004f94d128be: Pull complete
```

We can run the image using the code "`sudo docker run dwhitena/mgmt-validate:slim`"

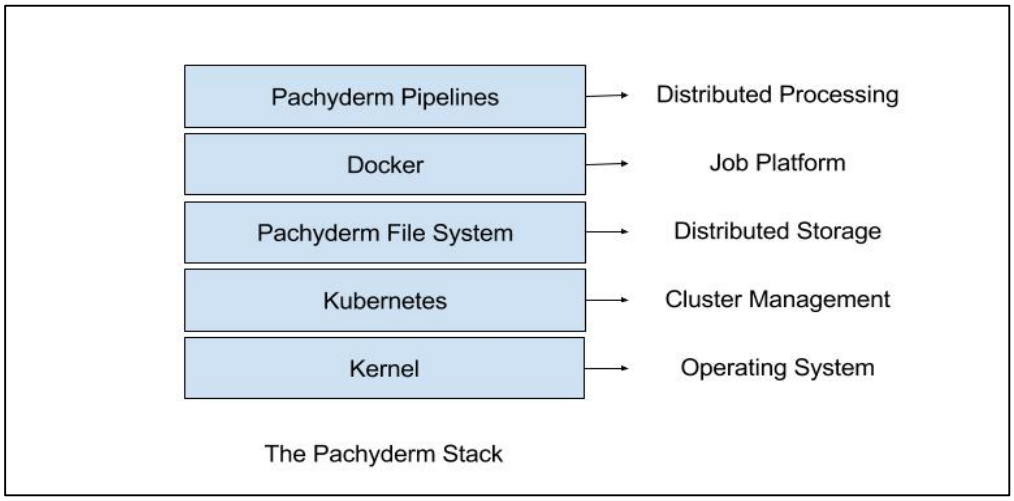
We can run it interactively using the code "`sudo docker run -it dwhitena/mgmt-validate:slim /bin/ash`"

This allows us to make changes in the container even when the Docker container is running the image. We could write codes to create directories called valid and invalid and jpeg image file called blah using "`/code # mkdir valid , /code # mkdir invalid , /code # touch blah.jpeg`"

We copied Jupyter notebook image file using "`sudo docker pull jupyter/notebook`" command and connected the container's output port at 8888 to Jupyter's port with the command "`sudo docker run -d -p 8888:8888 jupyter/notebook`"

5. **What will be different in the actual Pipeline, Kubernetes + Pachyderm**

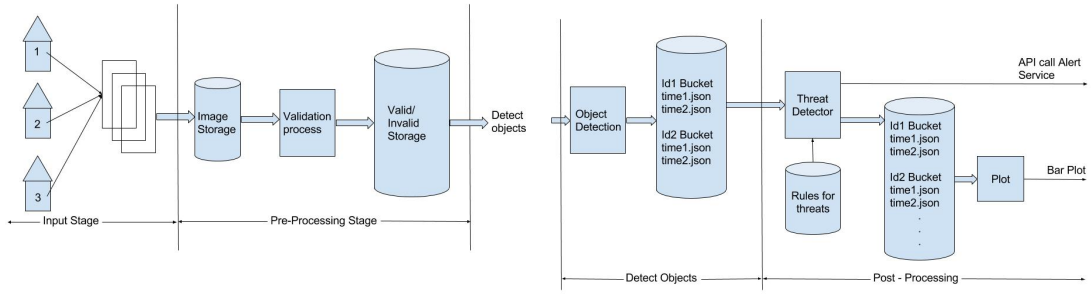
When we deploy many containers together we need to figure out an optimized way to run those containers. The right code needs to be run on the right input data at the right time. So, we need to automate the process of running code like we did in class. To do this we will have a Pachyderm on top of Kubernetes. The Pachyderm stack will look as follows:



The Pachyderm Stack

The diagram above shows how the Pachyderm stack works. Pachyderm pipeline is fed with image. Pachyderm will under the hood deploy the Docker container. Kubernetes knows which container should go where. Pachyderm gets data from data repositories and processes the input images by creating pipelines. Cluster management is becoming an important task as developers and businesses increasingly develop and deploy distributed applications in the cloud. Cluster management systems schedule work and manage the state of each cluster resource. We will use

Kubernetes to schedule the placement of containers across clusters based on which job needs to performed first and where the resources will come from. The Pachyderm stack will operate on Amazon EC2. This will cover the ‘image’ validation, object detection and threat detection stage of our pipeline.



In next step, we will integrate Docker, Pachyderm and Kubernetes together on EC2 while using Amazon cloud storage S3 to store the results of our analysis. This file will be stored as object in S3 and used for Post-Processing as discussed before.