# Docker Deployment System Built On

# Amazon Web Services

## Chris Fortier

## CSCI E-90 Cloud Computing

## Harvard University Extension School

## December 2014

# Project Description:

Docker and its associated ecosystem is one of the most hyped about technologies in the computing world. However many of the examples only show simple examples that may not be of practical use. In this project we will build a Virtual Private Cloud suitable for serving a web application using CloudFormation. We will then explore an innovative technique of running Docker containers on each server instance behind an HA Proxy process.

There are several important benefits to this approach. By running instances in an auto-scaling group we have elastic scalability to meet any amount of demand. By using HA Proxy on each host server, we will be able to deploy a new version of our application container to new visitors while simultaneously allowing all current connections to finish for current users. This affords the ability to do a rolling deployment without any visitor interruption.

# Table of Contents

Project Description: 2

Objectives: 4

Virtual Private Cloud 5

CloudFormation with Troposphere 5

Server Installation 7

Consul 7

Docker Hub 12

Boot2Docker 14

Build Application 14

HA Proxy 15

Deployment Process 16

Demo Initialization 17

Demo Deployment of B 18

Conclusion and Next Steps 19

References 20

# Summary

Docker and its associated ecosystem is one of the most hyped “new” technologies being discussed amongst the tech world. In this paper we will explore the basics of linux containers and Docker along with selected projects within the ecosystem.

The initial question explore is “what problem does Docker solve?”. Development of modern websites and cloud-based applications is usually down on various platforms. These platforms are typically very different between the development, testing, and production lifecycle and often times involve multiple operating systems and associated libraries. Docker attempts to solve this by packaging an application, its operating system, and all necessary dependencies into a container that can be deployed as a single unit.

The primary benefit afforded by this technology is that this singular container can be built once then moved through the pipeline from development to testing and finally to deployment. As with most technology, there are also some drawbacks. One initial concern is that Docker the company is only about a year and a half old and the codebase is still under extremely active development. This leads to concerns with stability and the future roadmap of the ecosystem.

As a demonstration we will explore a custom written deployment system designed to allow a host server to serve multiple versions of the same web application simultaneously. In this example we will provision the host with Consul, a distributed key/value store and service delivery system, to communicate configuration values. We will use HA Proxy on the host server to manage serving traffic to the Docker containers. This will allow us to deploy a new version of the application that will be visible to all new connections while allowing existing connections to complete their current session. Finally we will walk through a control program to make this all happen.

# Objectives:

This goal of this project is to demonstrate how to build a full-scale infrastructure on Amazon Web Services (AWS) that would be suitable for running a production web application. As the primary focus is to discuss Docker, I will make assumptions that the reader is already well versed in all of the underlying AWS technologies. Specific technologies include: Virtual Private Cloud (VPC), Auto-Scaling Groups, Elastic Load Balancers (ELB), Route53, Elastic Compute Cloud (EC2), and CloudFormation.

All source code will be made available on github.com and we will only use open source tools as needed.

# Virtual Private Cloud

We need to have to infrastructure in place to begin our demonstration. For maximum scalability and fault tolerance we will build a VPC with the following requirements:

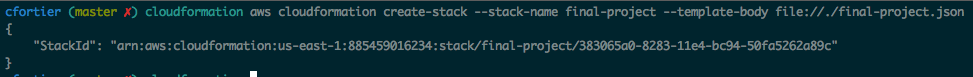
* (3) Availability Zones
* Auto-scaling group of “web-tier” instances spanning all three availability zones
* Auto-scaling group Consul server instances spanning all three availability zones
* Elastic Load Balancer to balance traffic to the auto scale group
* Security Groups allowing access for:
  + HTTP traffic from all addresses to the ELB
  + All HTTP traffic from the ELB to the auto scale instances
  + Traffic between all instances for our distributed key/value store

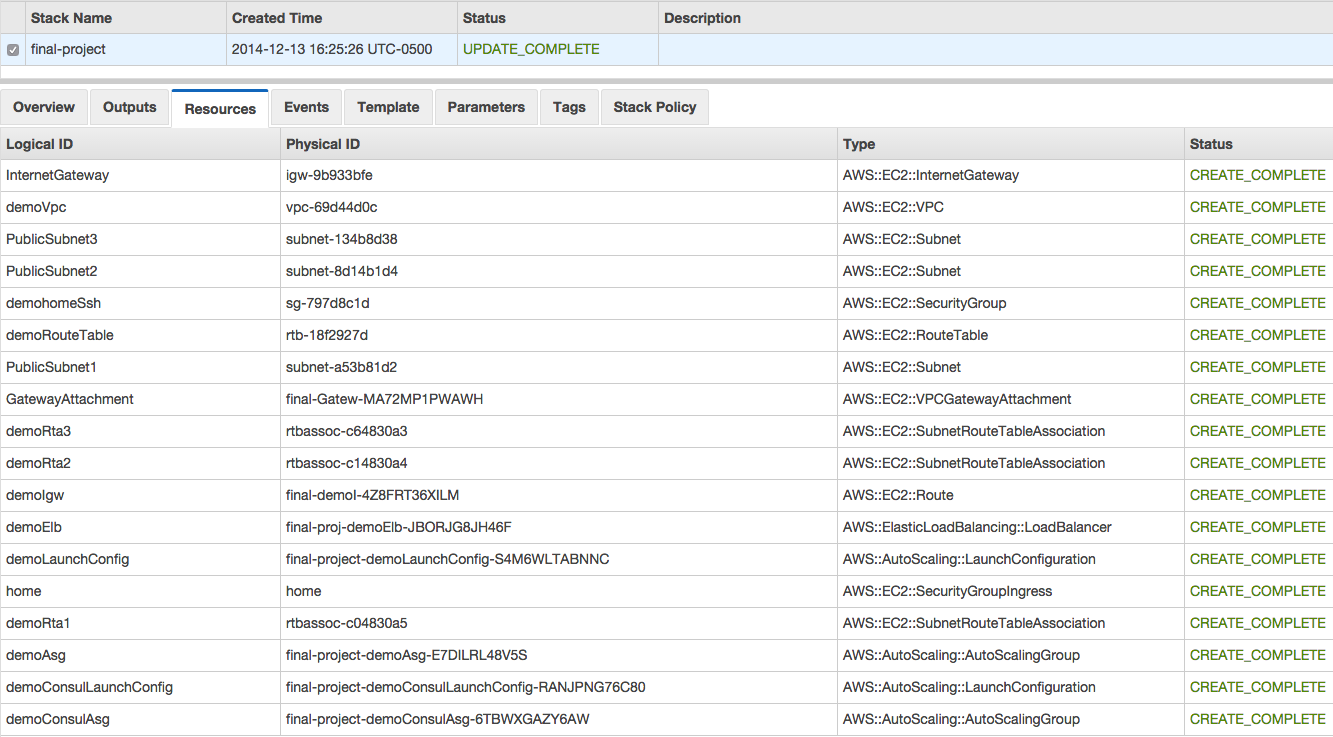
# CloudFormation with Troposphere

To build this VPC we will use an open source tool called *Troposphere* (<https://github.com/cloudtools/troposphere>), which is a Python library to automate the creation of CloudFormation templates. The source code used can be found at (<https://github.com/cfortier2/cs90-final/source_code/cloudformation>).

The Troposphere file is written to generate a CloudFormation template file named `final-project.json` in the same directory. Once the CloudFormation template is generated, the stack can be launched using the AWS command line tools.

Example:





# Server Installation

We are using base Ubuntu:14.04 images. We will need to install several programs for this to work correctly. The `install\_all.sh` can be run on all three Consul Servers and Demo Host instances to install everything. These scripts are located in the `source\_code/scripts/setup` directory.

# Consul

In a distributed system one of the challenges is to make sure each node has the correct configuration values to operate. One option is to have a central command server that you can use to `ssh` in to each server and write configuration files when they change. This may work fine for a small number of fixed servers but isn’t particularly efficient for a large number of servers, especially when they reside in an auto scale group and are rather ephemeral. For this example we will use a distributed key/value store called Consul (<https://consul.io>).

As described on its own site, “Consul has multiple components, but as a whole, it is a tool for discovering and configuring services in your infrastructure. It provides several key features: Service Discovery, Health Checking, Key/Value Store, Multi Datacenter”[[1]](#endnote-1) In the scope of this demonstration we will only be concerned with the key/value store, however the service discovery and health checking are vital components for any true production system.

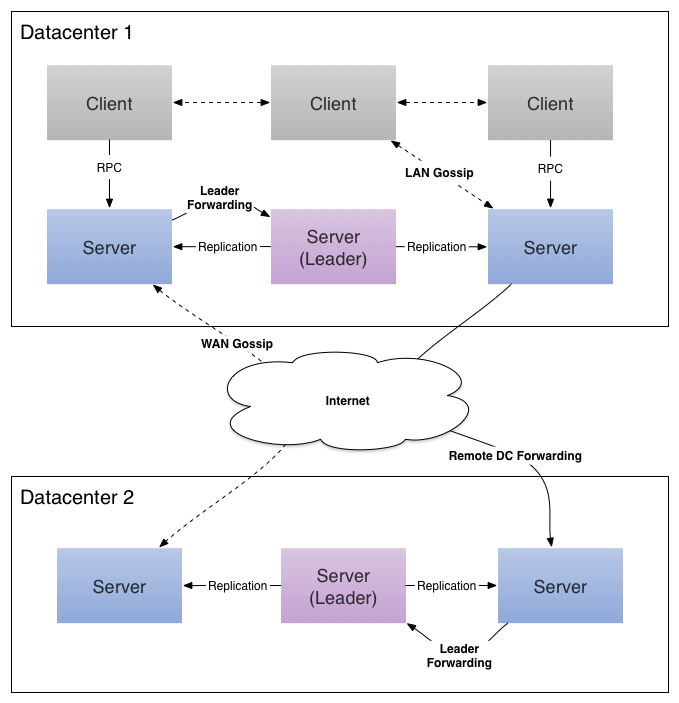
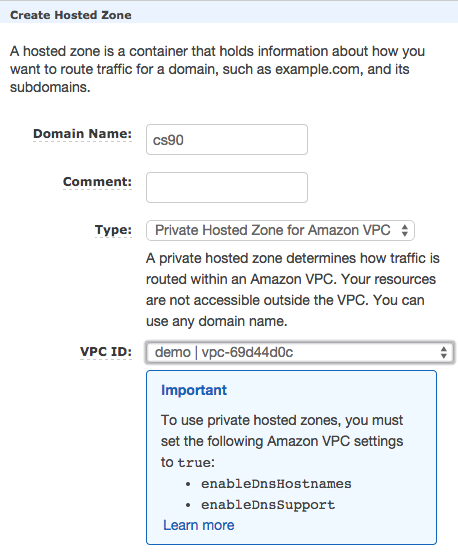
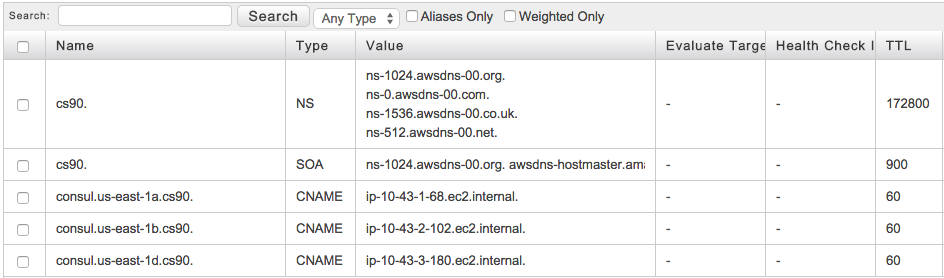


Figure Consul Architecture[[2]](#endnote-2)

Above is a diagram showing the Consul architecture for a multi datacenter deployment. For this demonstration we will have a cluster of three Consul Servers that are separate from the three client servers. Since we are bootstrapping a Consul server cluster we need a convention to locate all of them. The convention we will use is to create a Route53 Private DNS zone named `.cs90` then create a `CNAME` record based on the availability zone that the server resides in. Specifically, we will create: `consul.us-east-1a.cs90, consul.us-east-1b.cs90, consul.us-east-1d.cs90`. Since private DNS hosted zones are a new feature we will have to create the zone manually. The record creation is handled by: (<https://github.com/cfortier2/cs90-final/blob/master/source_code/scripts/route53.py>)

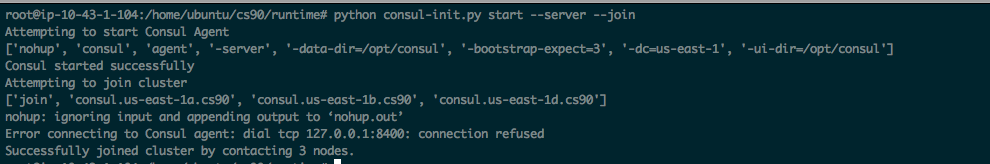


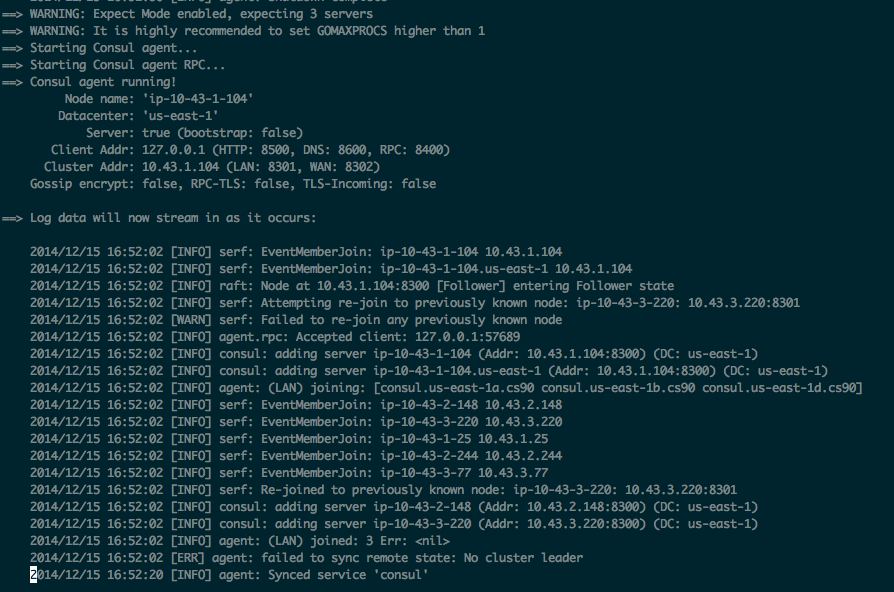


To install Consul I created a simple bash script, which can be found at: (<https://github.com/cfortier2/cs90-final/blob/master/source_code/scripts/install_consul.sh>). I also created a Consul init script which can is located at: (<https://github.com/cfortier2/cs90-final/blob/master/source_code/scripts/consul-init.py>). This init script is currently written to be run manually but should be configured to launch as an init.d or systemd type script.

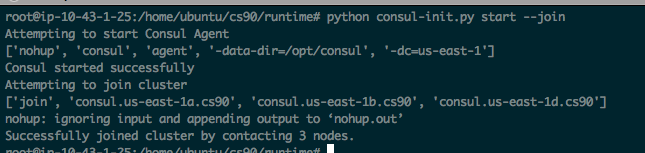
We need to initialize the three servers and also the three hosts. The same script can be run for both types of servers, but with different values. Starting the servers is accomplished by running `python consul-init.py start --server --join`. This will start consul in server mode and attempt to join the cluster.

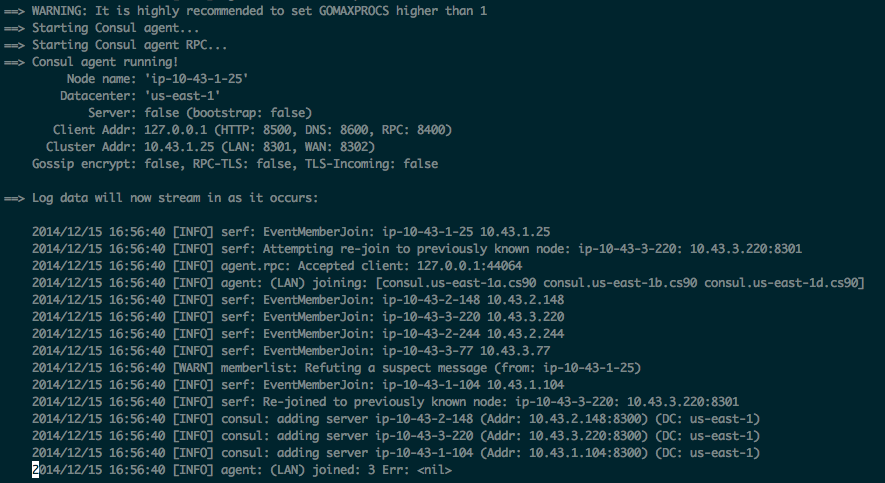
Example output from one of the servers produces:





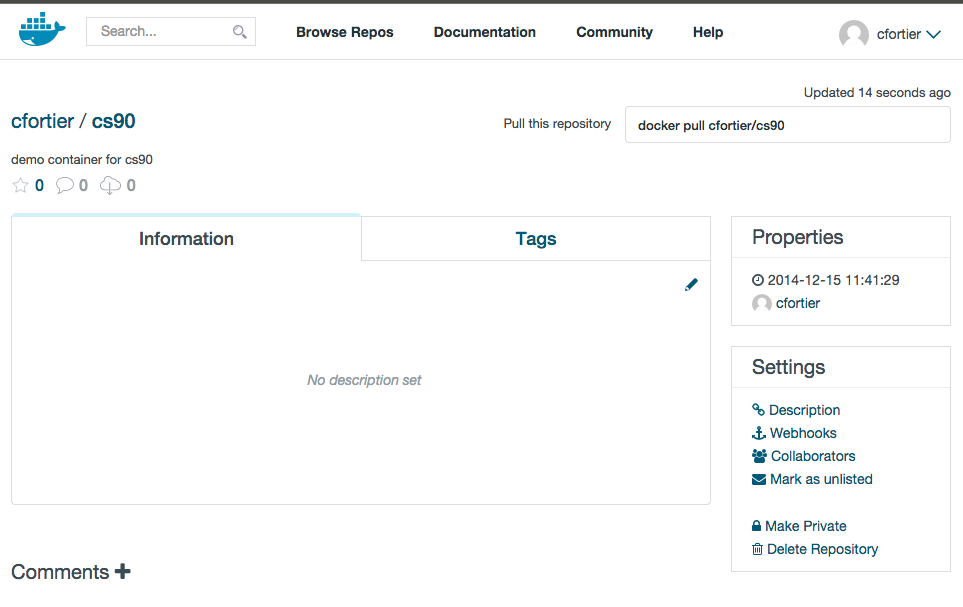
The last step to setting up the Consul cluster is to join the host nodes to the Consul cluster. This time we run the same command without the server option: `python consul-init.py start --server --join`. The output should show:





# Docker Hub[[3]](#endnote-3)

To work with Docker images we need a central location to store them. The open source solution is to use the Docker Hub, however many enterprise customers may want to consider any number of alternatives that provide private storage. Docker Hub operates very similar to github.com and uses very similar commands. For this demo we need to create a repository for our image:



Dockerfile**[[4]](#endnote-4)**

Now that we have a registry, we can work on building a Docker image. Dockerfiles are the essential building declaration of a Docker container. In our example we are going to build a simple container running the Apache server with a PHP application. Containers are designed to be composable and build upon the open source community. We will start with an official open source image for PHP: (<https://registry.hub.docker.com/_/php/>). Our Dockerfile will reside at (<https://github.com/cfortier2/cs90-final/source_code/docker/Dockerfile>).

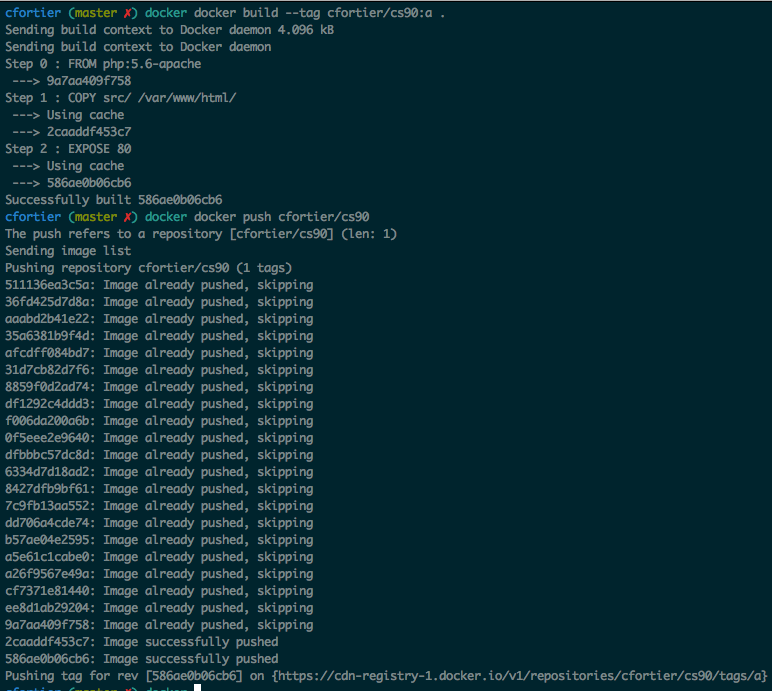
To give a quick overview of the Dockerfile: the first line states: `FROM php:5.6-apache` which is telling Docker to get the PHP image that is tagged with `5.6-apache`. The next line will copy the entire `src` folder in to the image as `/var/www/html`. The final line will expose port 80 on the container.

# Boot2Docker

Since we are developing on a Mac OSX machine, we need to build the Docker container somehow. This is accomplished by running a Linux virtual machine on the Mac. Boot2docker (<http://boot2docker.io/>) is an open source project dedicated to solving this problem. They provide all the tools necessary to run Docker on a Mac.

# Build Application

At this stage, we are finally ready to build our application. The application will consist of one PHP page that displays the host name and time. This is important to test the HA Proxy setup. The build and push output is:



# HA Proxy[[5]](#endnote-5)

HAProxy is a free, ***very*** fast and reliable solution offering [high availability](http://en.wikipedia.org/wiki/High_availability), [load balancing](http://en.wikipedia.org/wiki/Load_balancer), and proxying for TCP and HTTP-based applications. In our application we will run HA Proxy on each instance to serve traffic on port 80. HA Proxy will be configured at run time to know which Docker containers should be enabled and which ones will be disabled. The benefit of this is that a server can be placed in maintenance mode in HA Proxy while it is serving traffic. This has the effect of allowing any current connection to continue but will not server new traffic to it. This gives us the ability to deploy the new container while simultaneously running the current container until its transactions finish.

# Deployment Process

The process of deployment is rather detailed and needs to be designed to avoid any race conditions. With that in mind, we will view a deployment as a single transaction and will wait for all nodes to complete the transaction. For this demonstration we will manually handle this, however it should be done programmatically for an actual production system.

The process to initialize a node (on first boot) will be:

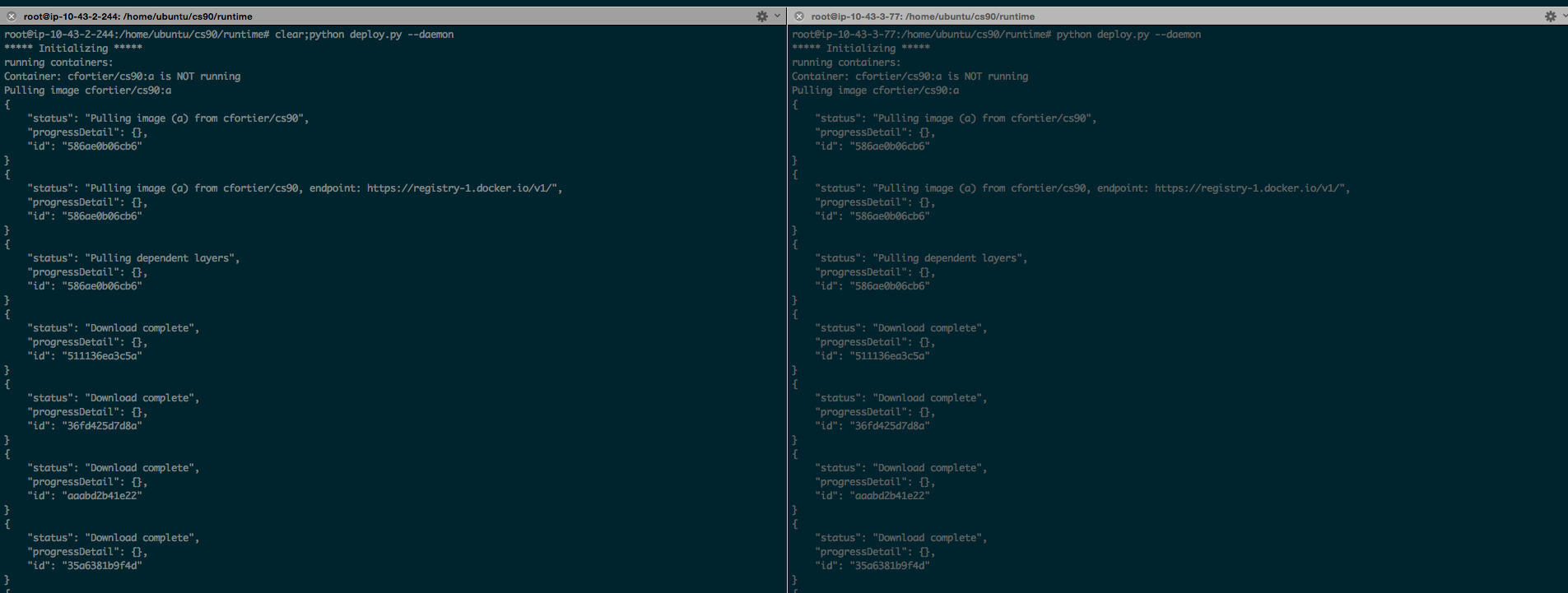
1. Query Consul `cs90/active/image` for the active Docker image
2. Pull that image from the Docker Registry
3. Run that image
4. Update HA Proxy to serve traffic to that node

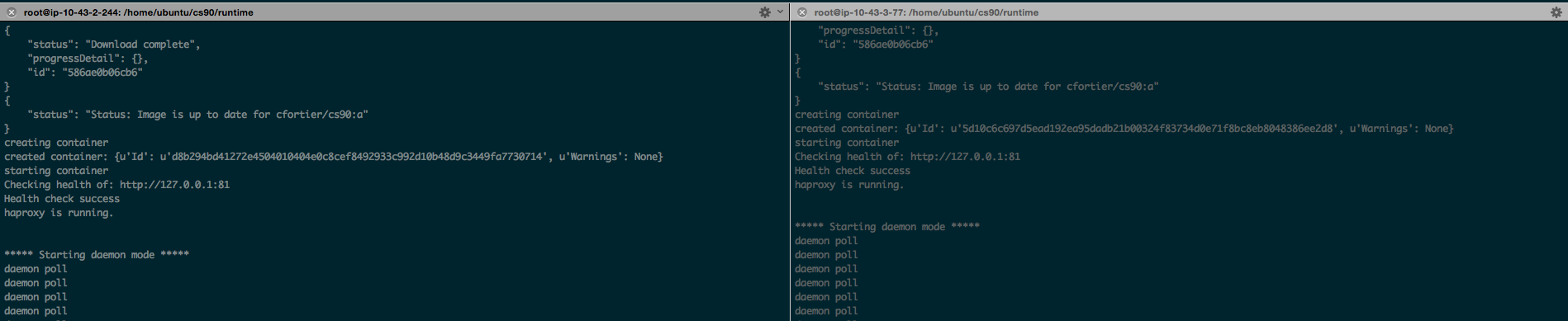
The process to deploy a new version to a running node will be:

1. Poll Consul every second for changes to the `cs90/stage/image`
2. If a change is detected, pull and run the designated image
3. Wait for the `cs90/active/image` value to be updated to the stage value
4. Update HA Proxy to server traffic to the new container while placing the original in maintenance mode
5. End transaction

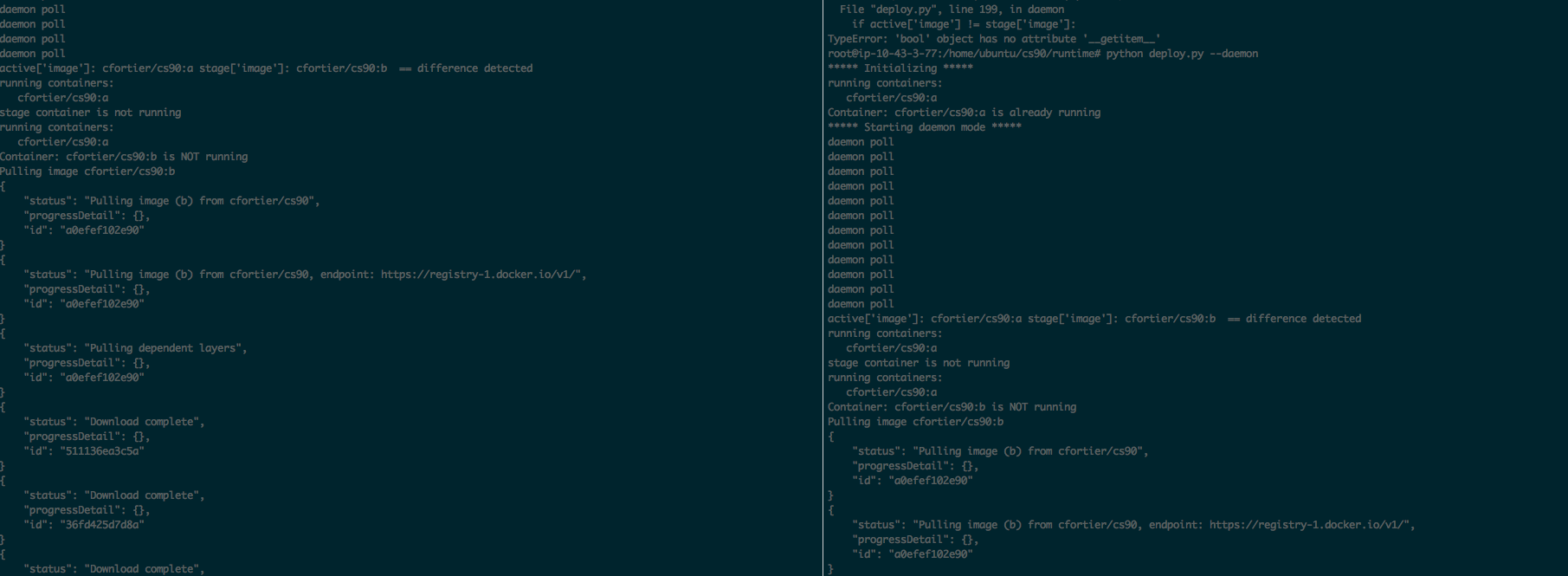
In the following screenshots we will go through an initialization and then a deployment. The top panel is one of the cluster members and will be used to update the Consul values. The two bottom panels are host nodes.

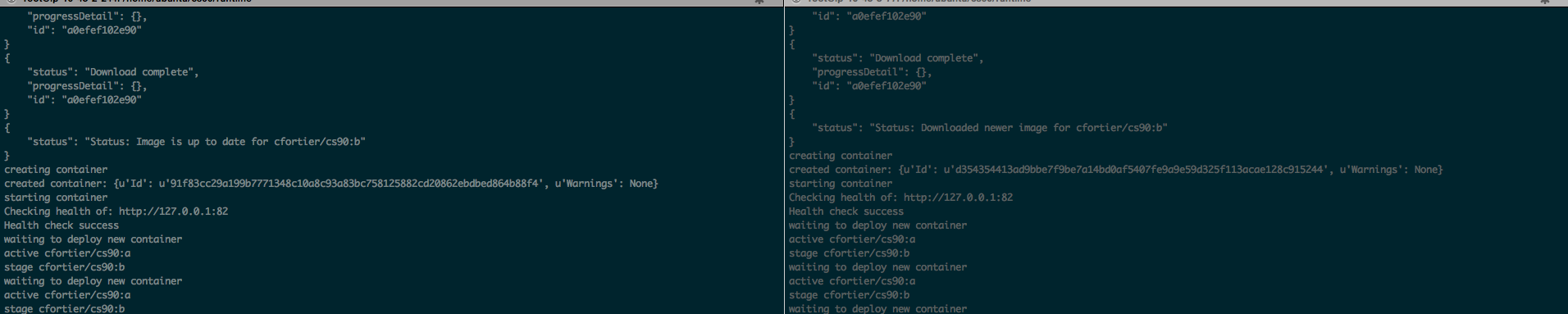
## Demo Initialization

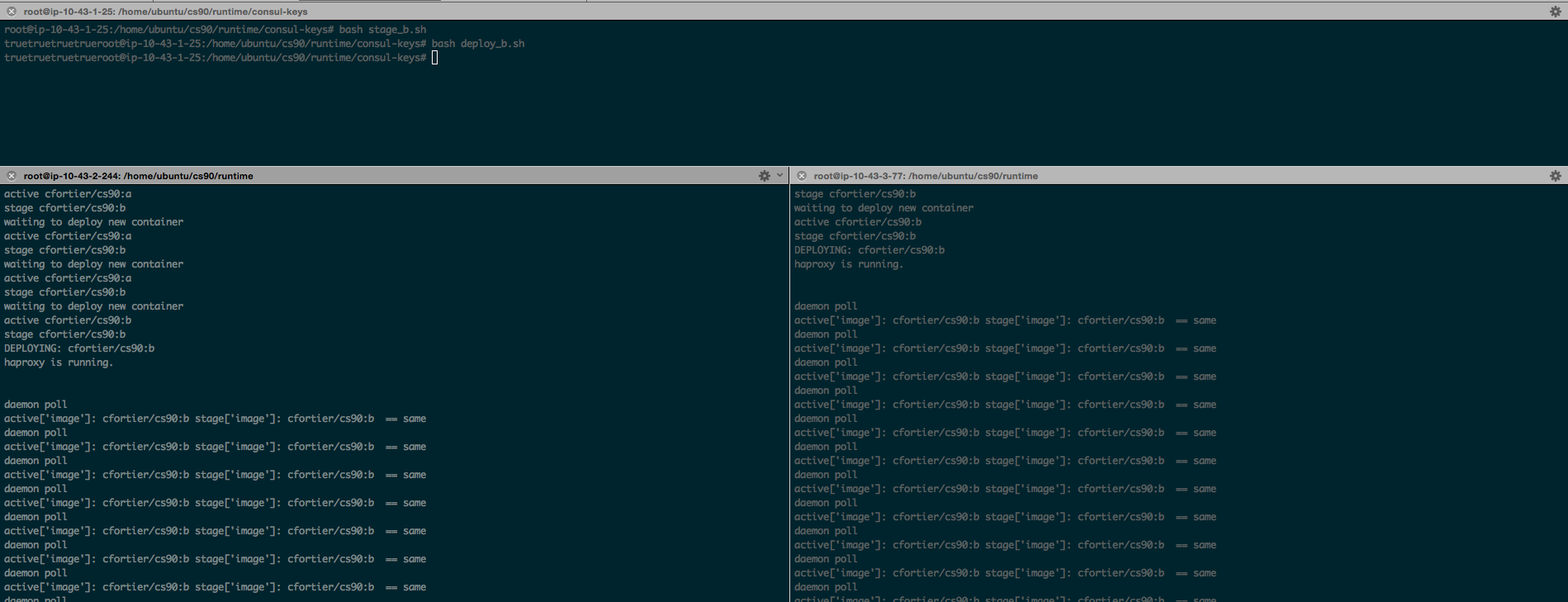




## Demo Deployment of B









# Conclusion and Next Steps

This paper has covered a wide range of topics but effectively shows how to build an entire AWS infrastructure from scratch, build a Docker Image, then execute a rolling deploy of those containers. There are several areas of improvement needed before this process would be considered “production ready”. Specific things that would need to be implemented:

1. Each node should report itself as providing a service in Consul. This would allow for real-time monitoring of the status of each node and would allow us to detect inconsistencies.
2. A command and control center would need to be implemented to execute all key updates (as opposed to the manual bash scripts we used).
3. Some mechanism would need to be implemented if the runtime scripts fail for some reason. Though it isn’t documented in pictures here, I encountered some random issues that caused a script to fail. However re-running the script worked fine.

# References

1. https://consul.io/intro/index.html [↑](#endnote-ref-1)
2. https://consul.io/docs/internals/architecture.html [↑](#endnote-ref-2)
3. https://docs.docker.com/userguide/dockerhub/ [↑](#endnote-ref-3)
4. https://docs.docker.com/reference/builder/ [↑](#endnote-ref-4)
5. http://www.haproxy.org/ [↑](#endnote-ref-5)