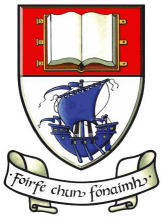

Waterford Institute of Technology

Cloud Technologies

The Automation of Infrastructure Orchestration and Application Deployment using Ansible and Docker Swarm

Stephen Coady

20064122

BSc in Applied Computing

November 29, 2016

Contents

1 Introduction

Modern applications are becoming increasingly complex, meaning it can also be complex to deploy the application. This research paper will examine application deployment, and will aim to show how modern tools and technologies can be used to simplify the process of building and deploying an application to the cloud.

It will compare these tools with "legacy" processes, evaluating the strengths and weaknesses of both. It will do this under the premise of a problem domain, discussed in Section 2.

Not only is application deployment a problem, but provisioning the servers which the application is hosted on is also something which needs to be considered. This paper will look at the automation of this process.

2 Problem Domain

One fitting definition of DevOps is "*DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support*" (Mueller et al., 2016).

Here the "service lifecycle" is the key term, as it means everything involved in creating a service and making it available for use in production. In older legacy systems the process would have involved separate teams performing each task along this timeline separately and without much overlap of personnel. In a simplified version of this model, the developers would write the code, the testers would test it, and the operations personnel would then deploy it along with provisioning the servers it would be deployed on.

This model however has shifted. In a survey conducted by (Logan, 2014), it is shown that, in general, DevOps orientated teams spend slightly more time on *all* tasks, whereas traditional IT roles will focus more on their primary tasks. This puts emphasis on the fact that developers now need to be more competent at multiple disciplines within IT. This, paired with the fact that the survey also shows DevOps orientated teams tend to automate more, shows that the importance of automated deployments has increased in recent times.

Also, with the rise in popularity of containers as a deployment vehicle, as can be seen for Docker specifically in Figure 1 it is more and more important to make the deployment of complicated applications more streamlined and reproducible.

To this end, this paper will aim to propose a solution to the problem of provisioning a production server and then deploying a fault-tolerant, scalable application to the server. It will try to show how

the process of building infrastructure and deploying an application to that infrastructure can be automated, increasing productivity and also allowing for an easily reproducible environment.

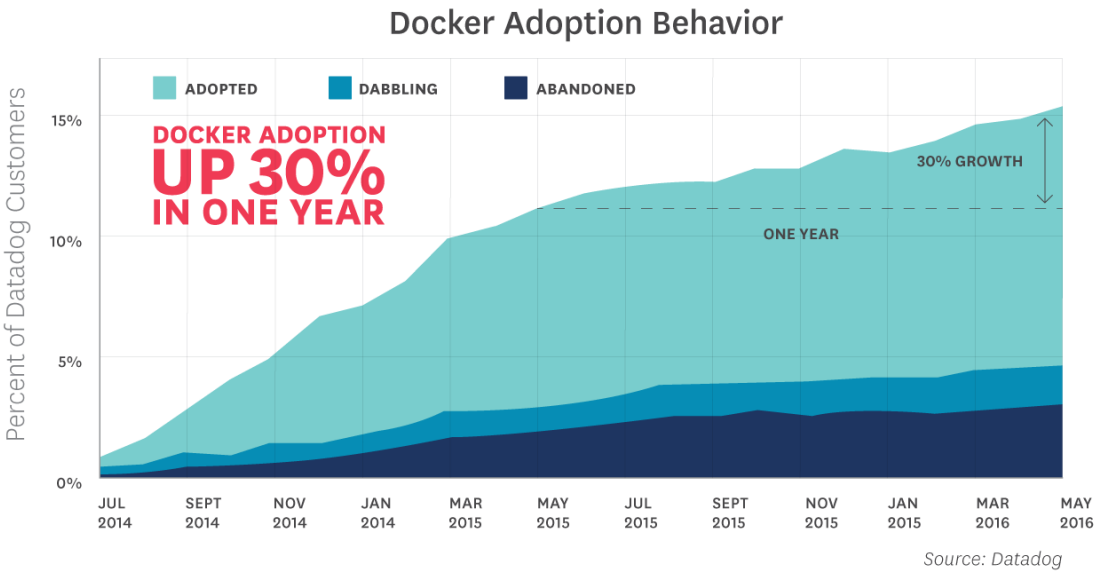

Figure 1: *Rise in Docker Usage. Credit: (DataDog, 2015)*

3 Technology Background

In this chapter we will examine a number of technologies that together will aim to solve the problem discussed in Section 2.

3.1 Containers

In software, a container is a process isolated from the host which generally runs in a very lightweight wrapper. It uses the underlying Linux kernel to do the work it needs to but ultimately it is a separate process which is self-contained (Matthias and Kane, 2015). To achieve this it uses the Linux abstraction method of namespaces and cgroups. Namespaces essentially present a segment of a global resource (such as processes, users, network) to a container and make the container think it has access to the global resource (Kerrisk, 2013). It is because of this that they can contain complete filesystems which can house everything the packaged application needs to run, including code, environment variables, libraries and dependencies while never actually having access to the Linux system it is running on.

Containers are often compared with virtual machines, however this view is a bit simplistic. Virtual machines are fully fledged operating systems running on a hypervisor which emulates dedicated hardware. It is made up of virtual devices which emulate the physical devices of a real host. Containers however, are not as full featured. Instead, they can be made so that they only have those resources that they need - and nothing else. So while virtual machines are emulating a real server, it could be said that a container is emulating a single *process* on a server - and only packaging exactly what that process needs to run.

Advantages
Some advantages of containers are:

- Lightweight - images can be as small as 5 MB
- Ephemeral - containers lifespan can be as small as is needed, they can be started to perform a single process and then stop as soon as they are done.
- Cheap - it is not CPU intensive to start a container
- Portable - containers can be built from a single file
- Secure - Using containers ensures applications are isolated from each other. An added benefit here is that multiple versions of the same application be running on the same host.

Disadvantages

There can also be drawbacks to containers, these include:

- Potential for added work - while containers can be useful they can also add to the work load and increase the lifecycle management of the application infrastructure.
- Orchestrating large applications can be complex - more moving parts
- The containers share the same kernel. Any issues with the kernel and the container engine running on it will affect all containers.

3.2 Docker

Docker is a software package which aims to orchestrate and manage containers for the user (Docker, 2016*e*). Docker uses its own driver called libcontainer to manage namespaces, cgroups and other Linux tools (Hykes, 2014) which in turn allow for containerization on a host. An illustration of this can be seen in Figure 2.

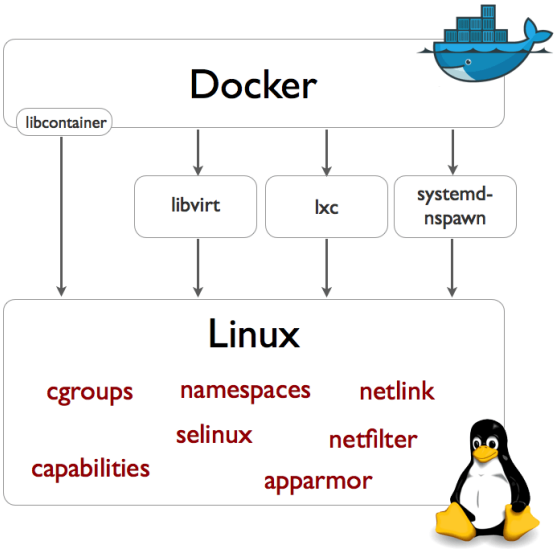

Figure 2: *Docker Using libcontainer. Credit: (Hykes, 2014)*

Docker allows for easily building, packaging and sharing of its "Docker images". It does this using two main components. The first component is the server-side program, Docker daemon. This is the part of the program which carries out the tasks required to run containers. The second component is the control over this daemon, the client. It can be accessed either via the API which Docker runs by default or through the command line interface (Docker, 2016*a*).

A Docker image is basically a set of instructions to create a template from which a container is run. The advantage of this is that multiple containers can be run from the same base image, meaning a reduction in storage size and making everything reusable.

One of Docker's strengths is the ease with which these images can be shared on used by anybody who wishes. To achieve this, Docker uses a shared repository model. Which means multiple users can 'push' images they want to use later to a remote server which can then be pulled later on multiple servers and run at will. To make this even more accessible Docker has its own storage solution, Docker Hub (Docker, 2016*b*). This website makes it easy for anybody to push and pull their own images or find new community-built ones to use themselves.

3.2.1 Swarm

While placing an application in a container makes it trivial to start an application and run it in a cloud service, it also introduces many new challenges. Two new technical challenges which this paper is interested in are:

- How many containers do we need to individually manage to run our application and across how many hosts?
- What happens when the application needs to scale?

To combat these problems Docker has produced an inbuilt tool called Docker Swarm (Docker, 2016*c*). Docker swarm aims to provide a mechanism to automatically manage a group of containers running across multiple remote hosts. In other words, "*It turns a pool of Docker hosts into a single, virtual Docker host*" (Docker, 2016*c*).

Docker swarm's aim is to allow the user to run an application within a fault-tolerant, scalable and easily reproducible system. We will look further at how swarm works in section 4.

3.2.2 Alternatives

There are many alternatives to Docker. When Docker first started, instead of using libcontainer as discussed in section 3.2 Docker actually relied completely on a project called Linux Containers (LXC). LXC can be seen in Figure 2 also. However LXC is also a standalone project, enabling users to use container technologies. LXC is currently transitioning to LXD, which is a command line tool and API to interact with the program (LXC, 2016).

Another competing product, which happens to focus on containers at scale is an open source tool to allow for orchestration of large container-based applications (Kubernetes, 2016). This is a direct competitor with Docker swarm, offering portable, extensible and self-healing container pools.

There are certain disadvantages to using both of these tools however. With LXC, it is an older technology but the community is not as strong. This means that it does not come with as many images to use. Kubernetes is a strong clustering tool, however the pure volume of Docker images available again means it is probably a better place to start. Kubernetes is also completely aimed at running a cluster or, as they call them, pods (Kubernetes, 2016). Whereas Docker is aimed at single container applications if necessary and then Docker swarm allows for clustering.

3.3 Ansible

Ansible is a tool to manage a server using ssh (Ansible, 2016). The server can be either remote or local, once it is available via ssh. The main advantages of Ansible are that it is:

- Agentless - requires no client installed on the machine which needs to be managed.
- Idempotent - can be safely run multiple times and if no change is required then a change will not occur.
- Simple - with Ansible there is a low barrier to entry as all files are written in YAML.
- Powerful - it can be used to manage a single server or one thousand.

Ansible also uses a file known as an inventory to manage the servers it can connect to. This is useful as it means servers can be grouped by name, location, application etc and Ansible can be instructed to only carry out certain tasks on certain groups of servers depending on our needs. We will look at this further in section 4.

To name just a small subset of Ansible's use cases:

3.3.1 Provisioning

The term provisioning means the setting up of new servers with which you will later interact (Hochstein, 2013). In other words, if you start with a blank cloud storage provider running no servers then to provision what servers you need is to create however many virtual machines are needed. Ansible has a number of modules available here, which allow Ansible to talk to third party providers such as Amazon EC2, Microsoft Azure etc.

We can see a simple example of this in listing 1.

Listing 1: Playbook To Create Instances On EC2

```
---
- hosts: localhost
  connection: local
  gather_facts: false
  user: root
  tasks:
  - name: Provision EC2 Nodes
    local_action:
      module: ec2
      key_name: "key.pem"
      group_id: "security_group_id"
      instance_type: "t2-micro"
      image: "some_AMI"
      vpc_subnet_id: "some_subnet"
      region: "some_region"
      assign_public_ip: yes
      count: "1"
---
```

Listing 2: A Simple Playbook To Deploy and Run Code

```
---
- hosts: servers
  gather_facts: false
  sudo: true
  tasks:
  - name: Pull sources from the repository.
    git: repo={{ project_repo }} dest={{ project_root }}/code/
  - name: Start Application
    command: python /code/app.py
---
```

3.3.2 Configuration Management

Configuration management is described as *"...writing some kind of state description for our servers, and then using a tool to enforce that the servers are, indeed, in that state..."* (Hochstein, 2013). This includes ensuring the correct software packages are installed, the correct services are running and any configuration files needed are present. Ansibles allows us to do this as, as previously stated it is idempotent. This means that if for some reason a server we wish to manage is only 50% percent set up we can safely run an Ansible playbook which performs the *complete* set up process and not worry that it will cause harm, i.e. Ansible will know it does not need to perform the first 50% of configuration again and will just inform us that no change was needed for these steps.

3.3.3 Deployment

Deployment can be defined as *"...the process of taking software that was written ... copying the required files to the server(s), and then starting up the services."* (Hochstein, 2013). In this regrard Ansible excels. Ansible uses a concept of playbooks to run tasks against remote servers, an example of which can be seen in listing 2.

In this code we are simply copying code from the git repository to the server and then starting the code from within that directory. Although this is a rather simplistic version of what Ansible can do it does show how easy it is to go from a bare server to one running an application.

3.3.4 Alternatives

While there are many other configuration management tools available such as Chef and Puppet, these both require an agent to be installed on the remote server being managed. This is a huge benefit of Ansible over other tools and so Ansible was chosen for this reason.

4 Building an Application

We will now look at building an application using the technologies previously discussed in section 3. This will involve several distinct steps, including:

- Provision instances which will host our application.
- Running a custom application in a Docker container.
- Configuring these instances depending on the role they are to play.
- Deploying the application to these instances
- Running an application in Docker Swarm mode
- Creating a Docker loadbalancer which will sit in front of the application instances on a separate instance

All of these steps will then be automated using Ansible. The complete source code used to carry out all of the steps is available in Appendix A.

4.1 Provisioning The Infrastructure

We will first look provisioning the servers needed for this application. The only dependency needed to run these playbooks in Ansible is the Boto library (The Boto Project, 2016) installed locally and also python. For the purpose of this report, the EC2 instances required will be split into 3 distinct groups. These are:

- Management
- Node
- Loadbalancer

We will discuss what these terms mean further in section 4.4, but for now we do not need to know the difference, just that all instances created have an intended role as our application orchestration continues.

To create this infrastructure we can uses Ansible's built in ec2 module, which makes it easy to run commands against AWS. We can see the playbook used to create 3 sets of ec2 instances in Listing 3. This playbook is the set of steps to follow but it relies on Ansible roles and variables to run, as seen in Listing 4.

Listing 3: create-instances.yml

```
---
# create manager nodes
- hosts: localhost
  connection: local
  gather_facts: false
  user: root
  pre_tasks:
    - include_vars: ~/dev/docker_with_ansible/ec2_vars/managers.yml
  roles:
    - ~/dev/docker_with_ansible/roles/provision-ec2-managers

# create worker nodes
- hosts: localhost
  connection: local
  gather_facts: false
  user: root
  pre_tasks:
    - include_vars: ~/dev/docker_with_ansible/ec2_vars/nodes.yml
  roles:
    - ~/dev/docker_with_ansible/roles/provision-ec2-nodes

# create the loadbalancer
- hosts: localhost
  connection: local
  gather_facts: false
  user: root
  pre_tasks:
    - include_vars: ~/dev/docker_with_ansible/ec2_vars/loadbalancer.yml
  roles:
    - ~/dev/docker_with_ansible/roles/create-ec2-loadbalancer
```

Listing 4: provision-ec2-managers.yml

```
---
 - name: Provision EC2 Managers
   local_action:
     module: ec2
     key_name: "{{ec2_keypair}}"
     group_id: "{{ec2_security_group}}"
     instance_type: "{{ec2_instance_type}}"
     image: "{{ec2_image}}"
     vpc_subnet_id: "{{ec2_subnet_id}}"
     region: "{{ec2_region}}"
     instance_tags: '{"Name":"{{ec2_tag_Name}}","Type":"{{ec2_tag_Type}}"}'
     assign_public_ip: yes
     wait: true
     count: "{{manager-count}}"
     volumes:
     - device_name: /dev/sda1
       device_type: gp2
       volume_size: "{{ec2_volume_size}}"
       delete_on_termination: true
   register: ec2
```

The role in Listing 4 is to provision management nodes, however the roles for regular worker nodes and loadbalancers are the exact same. We can see the use of curly braces - anything in a set of double curly braces are variables within Ansible. These are defined in a separate file and loaded into the role by using the key: `pre_tasks` as seen in Listing 3. These variables can be called from within Ansible and used to dynamically assign values. To run the playbook in Listing 3 we can now run the command:

```
ansible-playbook -e "manager-count=1" -e "worker-count=2"
-e "loadbalancer-count=1" playbooks/create-instances.yml
```

Here we use the variable flag -e to set the count of how many of each type of instance we require. This variable is then used in the relevant roles. Once we have run this command, we will now have the desired number of instances running in AWS.

4.2 Ansible as a Configuration Management Tool

Now that we have provisioned the number of instances we need we can proceed to make sure they are in the desired state. In other words, we can use Ansible as a configuration management tool to ensure the instances have everything they need to run the processes we will need.

The only configuration we require for this demonstration is that the correct version of Docker is installed on each instance. Since every instance in the application will use Docker to run its applications, this role will be run on all instances. This role can be seen in Listing 5.

To run this role against all of our nodes, we use the playbook in Listing 6.

This listing specifies that the role be run against managers, nodes and loadbalancers.

Listing 5: Role to install Docker on a Ubuntu Instance

```
---
# tasks file for docker-engine
- name: Ensure the system can use the HTTPS transport for APT
  stat:
    path: /usr/lib/apt/methods/https
  register: apt_https_transport

- name: Install HTTPS transport for APT
  apt:
    pkg: apt-transport-https
    state: installed
  when: not apt_https_transport.stat.exists

- name: Import Docker key into apt
  apt_key:
    keyserver: hkp://p80.pool.sks-keyservers.net:80
    id: 58118E89F3A912897C070ADBF76221572C52609D

- name: Add Docker deb repository
  apt_repository:
    repo: 'deb␣https://apt.dockerproject.org/repo␣ubuntu-trusty␣main'
    state: present
    update_cache: yes

- name: Install Docker Engine
  apt:
    pkg:
      - docker-engine={{docker_version}}*
    state: installed

- name: ensure docker runs without sudo
  command: usermod -aG docker ubuntu
---
```

Listing 6: install-docker-engine.yml

```
---
- hosts: managers:nodes:loadbalancers
  gather_facts: false
  become: true
  roles:
    - ~/dev/docker_with_ansible/roles/install-docker-engine
---
```

Listing 7: run-wordpress.yml

```
---
- hosts: node
  become: true
  tasks:
    - name: copy source code
      shell: >
        git clone https://github.com/nezhar/wordpress-docker-compose.git

    - name: change directory to code and run application
      shell: >
        cd wordpress-docker-compose && docker-compose up -d
```

4.3 Running an Application in Docker

Now that we have instances running and they are in the state we want them in we can see how easy it is to run an application on one of these instances.

To do this using the command line, we can simply run

```
docker run --name some-wordpress --link some-mysql:mysql -p 80:80 -d wordpress
```

which will start a Wordpress container, link it to a mysql container running on the host and expose port 80 of the container (the port the application is listening on from within the container) to port 80 of the host. However if we wish to automate this process and start an application we can use Ansible to do this for us.

In this step we will use a tool made by Docker called 'Docker Compose'. Docker compose allows us to run multi-container systems in one command line call, managing links between containers and all container dependencies and settings for us. In Listing 7 we can see that Ansible first copies source code we require from Github, which we then tell Ansible to run using Docker Compose.

The result of this simple playbook is that we now have a fully functional Wordpress installation running in AWS, complete with a MySQL database. We can see the Docker Compose file in Listing 8 and the resulting application in Figure 3.

Since this application is running on one instance in AWS it is currently not:

- Scalable
- Fault tolerant
- Distributed

To address these issues we will next look at Docker Swarm.

Listing 8: docker-compose.yml Credit: (Nezbeda, 2016)

```
---
version: '2'
services:
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80 # change ip if required
    volumes:
      - ./wp-app:/var/www/html
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: root
      WORDPRESS_DB_PASSWORD: password
    links:
      - db:db
    networks:
      - wordpress-network
  db:
    image: mysql:latest # or mariadb
    ports:
      - 3306:3306 # change ip if required
    volumes:
      - ./wp-data:/docker-entrypoint-initdb.d
    environment:
      MYSQL_DATABASE: wordpress
      MYSQL_ROOT_PASSWORD: password
    networks:
      - wordpress-network

networks:
  wordpress-network:
      driver: bridge
```

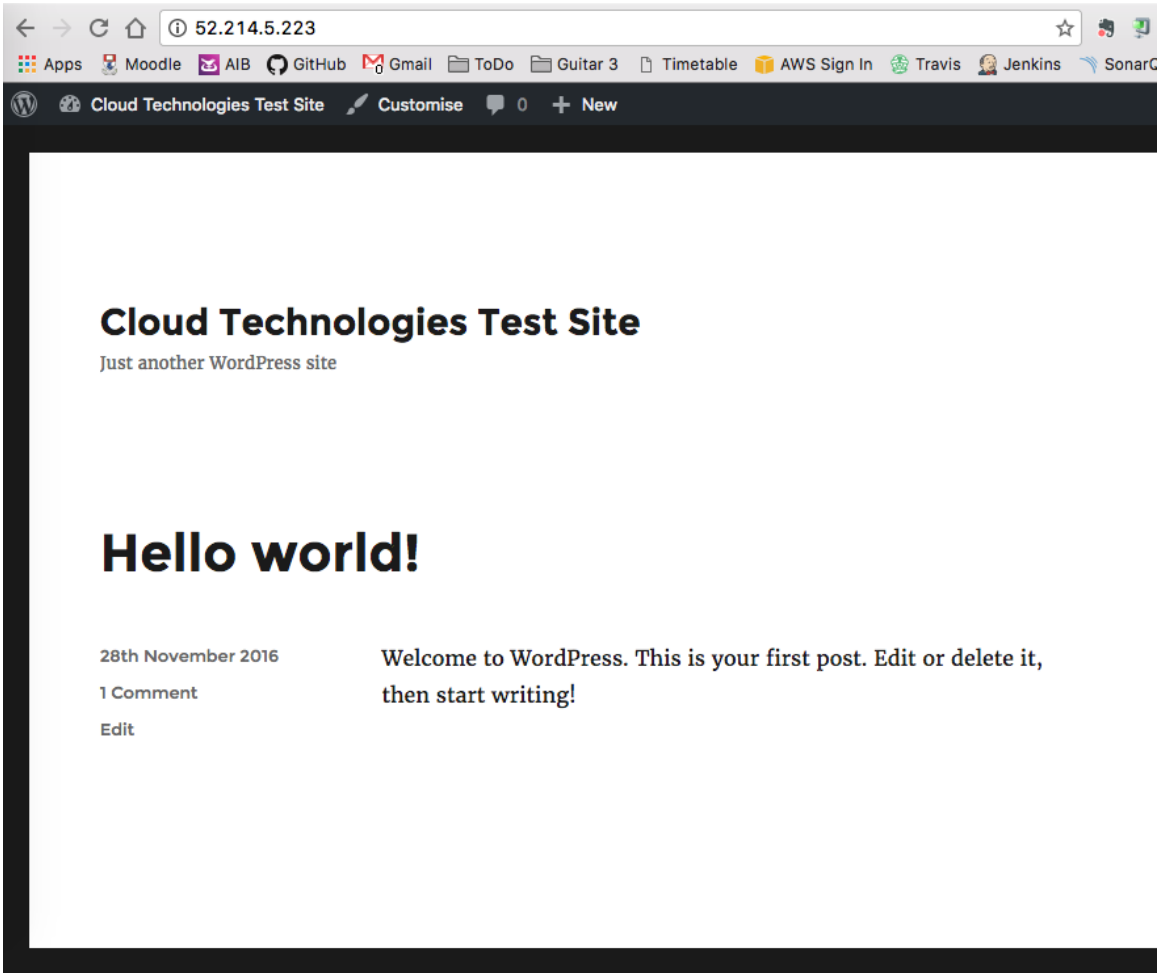

Figure 3: *Fully Functional Wordpress within a Container*

4.4 Creating the Swarm

To create a swarm we will use an integrated feature of Docker Engine (after version 1.12) called swarm mode (Docker, 2016*d*). Swarm mode requires a minimum of 1 instance, although this would be pointless. To prove beneficial, 2 instances must be used where at least one is in 'manager' mode. Any node in manager mode will act as the organiser of the swarm, receiving commands on behalf of the swarm and delegating tasks accordingly.

There are some pre-requisites to running swarm mode, mainly concerning port availability. These are:

- TCP port 2377 must be available on the manager instance

- TCP and UDP ports must be available on all swarm instances for communication
- TCP and UDP port 4789 must be available on all swarm instances for networking purposes

Once we have satisfied these conditions by creating our instances and adding them to an appropriate security group as seen in Listing 3 we can then initialise swarm mode on the instances. To do this, we first enter the command shown in Figure 4. This returns a token which we will use to join a node to the swarm.

```
ubuntu@ip-172-31-15-182:~$ docker swarm init --advertise-addr=172.31.15.182:2377
Swarm initialized: current node (b5dpqvcwzf4y868johfyhez7z) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-3t3xj5h7hb764psmhs8v3z94rs8z8376kq57xwf019qjzlnbpf-4dodrza7pev8tcx6guvwaa7w5 \
    172.31.15.182:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Figure 4: *Initialise Swarm Mode*

Once the manager is added we can now add nodes to the swarm. In the next section we will see how we can add multiple nodes but for the purpose of explaining the basics of the swarm we will add just one node for now. We can see in Figure 5 that to join a node to swarm mode we can enter the command given previously in Figure 4. This node is now part of this swarm and available on the manager node as can be seen in Figure 6.

```
ubuntu@ip-172-31-7-75:~$ docker swarm join \
>     --token SWMTKN-1-3t3xj5h7hb764psmhs8v3z94rs8z8376kq57xwf019qjzlnbpf-4dodrza7pev8tcx6guvwaa7w5 \
>     172.31.15.182:2377
This node joined a swarm as a worker.
```

Figure 5: *Join a Docker Swarm*

```
ubuntu@ip-172-31-15-182:~$ docker node ls
ID                            HOSTNAME          STATUS  AVAILABILITY  MANAGER STATUS
3x087grgk1haddj0vh22bhajv     ip-172-31-7-75    Ready   Active
b5dpqvcwzf4y868johfyhez7z *   ip-172-31-15-182  Ready   Active        Leader
```

Figure 6: *List All Nodes in a Swarm*

We can now start an application, called a service when in Docker swarm mode. To do this we run the command

```
docker service create --name app --replicas 2 -p 80:8080 nginx
```

which will create 2 replica containers in the swarm. We can then run docker service ps <service name> to inspect the service and see which nodes it is running on.

```
ubuntu@ip-172-31-15-182:~$ docker service ps app
ID                         NAME   IMAGE  NODE              DESIRED STATE  CURRENT STATE           ERROR
1lv2hbhztaoqkl25g0lve5yr2  app.1  nginx  ip-172-31-15-182  Running        Running 47 minutes ago
37rbqn5cql5djl4849yi80cu6  app.2  nginx  ip-172-31-7-75    Running        Running 47 minutes ago
```

Figure 7: *Listing the Service in the Swarm*

We can see in Figure 7 that our service is running on both instances, but we can also scale our service and watch as it is scaled as evenly as possible across our instances, as seen in Figure 8.

```
ubuntu@ip-172-31-15-182:~$ docker service scale app=4
app scaled to 4
ubuntu@ip-172-31-15-182:~$ docker service ps app
ID                         NAME   IMAGE  NODE              DESIRED STATE  CURRENT STATE           ERROR
1lv2hbhztaoqkl25g0lve5yr2  app.1  nginx  ip-172-31-15-182  Running        Running 52 minutes ago
37rbqn5cql5djl4849yi80cu6  app.2  nginx  ip-172-31-7-75    Running        Running 52 minutes ago
3o0f3ck2w07nwvpjw6s0thhow  app.3  nginx  ip-172-31-15-182  Running        Running 5 seconds ago
0yhriv7eiryy35up93pyczoap  app.4  nginx  ip-172-31-7-75    Running        Running 4 seconds ago
```

Figure 8: *Listing the Service in the Swarm After Scaling*

This process can also be automated using Ansible. To do this tasks in Listing 9 must be run against the appropriate hosts, the playbook is concatenated here for brevity.

Listing 9: create-and-join-swarm.yml

```
---
- name: initialize swarm cluster
  shell: >
    docker swarm init
    --advertise-addr={{ swarm_iface | default('eth0') }}:2377

- name: retrieve swarm manager token
  shell: docker swarm join-token -q manager
  register: swarm_manager_token

- name: retrieve swarm worker token
  shell: docker swarm join-token -q worker
  register: swarm_worker_token

- name: join worker nodes to cluster
  shell: >
    docker swarm join
    --advertise-addr={{ swarm_iface | default('eth0') }}:2377
    --token={{ swarm_worker_token }}
```

To better illustrate what is happening in the swarm an application a Docker swarm visualiser application was used (Marks, 2016). This application was run on a manager node and allows the user to graphically see which services are running on each node. To preview this application a swarm consisting of 5 nodes (2 managers and 3 workers) and several different services was started. This can be seen in Figure 9.

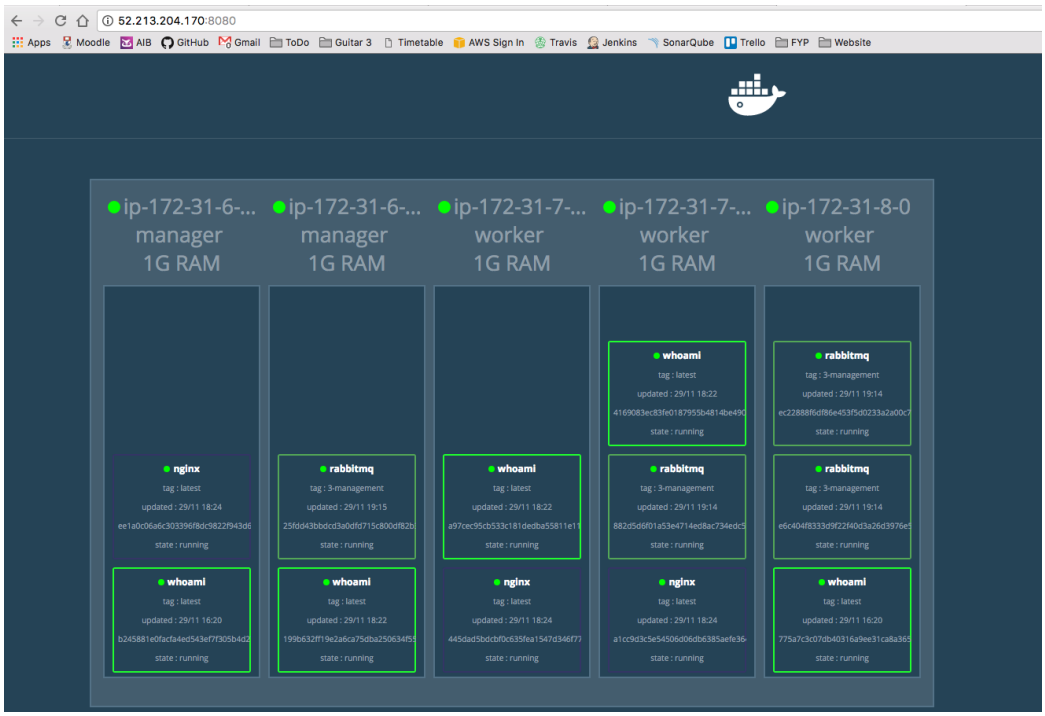

Figure 9: *Visualising the Swarm*

4.5 Load Balancing the Swarm

In the previous section we saw how Docker Swarm can provide resiliency by scaling an application across multiple nodes in a network. While this will allow the application to deal with a portion of the instances serving it to crash it does not make using the application any simpler. This is because there is no central point of contact to any container running within the swarm. To mitigate this a loadbalancer was created and placed in front of the swarm. This loadbalancer is then dynamically configured to rotate calls to different nodes in the swarm based on port assignments. An outline of this can be seen in Figure 10.

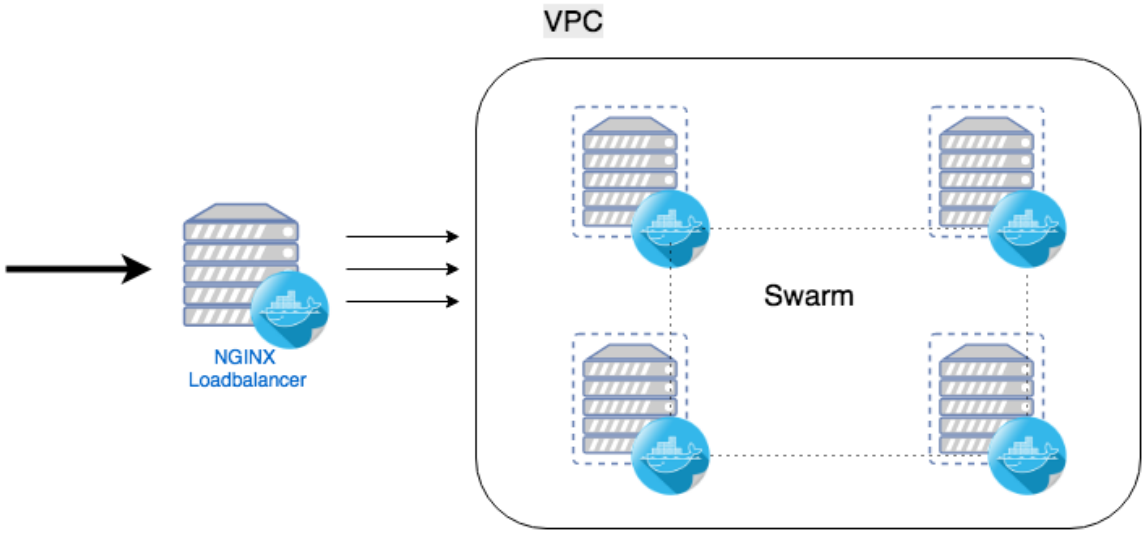

Figure 10: *Final Application Setup*

To achieve this with the swarm built in the previous section nginx was used as a loadbalancer and was also provisioned using Ansible. The same steps were followed as with the swarm manager and nodes in provisioning and configuration. This resulted in an AWS instance being deployed with Docker installed. Once this instance was ready, Ansible then dynamically created a config file which would enable an nginx instance to loadbalance between the hosts in this config file.

However since this paper is aimed at showing how Docker can improve applications and their deployment we will examine how nginx running within a container can benefit an application and its deployment. For this paper a Dockerfile to build the image was copied to the loadbalancer and was built using Ansible. In reality this could have been pulled from a Git repository using Ansible also. The steps involved in running the loadbalancer are:

1. Have Ansible create a dynamic configuration file which has all instances

2. Ansible then copies this config file to the loadbalancer
3. Ansible copies a basic Dockerfile to the loadbalancer (this contains a pointer to the previously copied config file)
4. Ansible instructs the host to build the Docker Image and run it listening on port 80

An overview of the result of this can be seen in Figure 11.

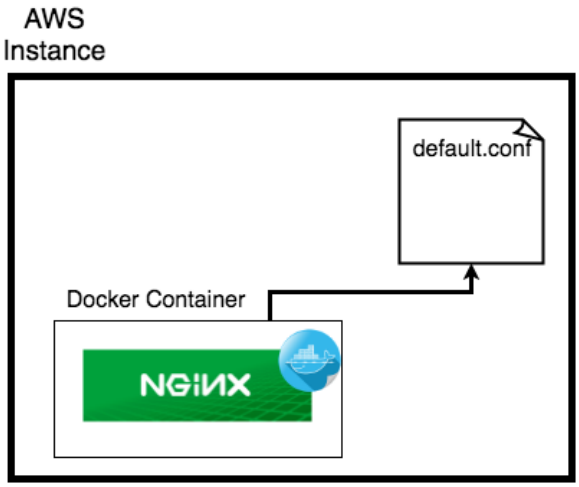

Figure 11: *Nginx Running in A Docker Container Acting as a Loadbalancer*

4.6 Features of Swarm

Now that we have seen how we can provision, configure and deploy an application using Ansible it is not hard to see how all of this can be tied together. The final product of applying everything from previous sections will mean that to deploy an application in swarm mode will consist of simply running an Ansible playbook. A separate Ansible playbook could then be created which will use existing inventory (saved by Ansible upon creation) to update the running application. In swarm mode, this might consist of running something similar to the command

```
docker service update --image nginx:3.0.7 nginx
```

Which will, in a rolling fashion, update each service using the nginx image to the image version 3.0.7. This ensures that the application does not need to be brought down to update.

Swarm mode also enables 'draining' of nodes within the swarm, which means removing all services running on a specific node and moving them to another node. This feature is useful when instances

need to be updated, say with security patches. Effectively we can remove and re-add instances one-by-one to the swarm using the command

```
docker node update --availability drain <NODE ID>
```

It is also important to note that while the application built is now being load balanced by an nginx Docker container it is also being load balanced *internally* by the swarm loadbalancer. This is extremely useful as it means the user does not need to worry about scaling an application and the effect doing this will have on the application running. One example of where scaling the number of containers running up can cause problems is when those containers are listening on certain ports. For example if we have an nginx container listening on port 80 on the host and we try to add a second container to provide redundancy then we will be greeted by a port conflict. In swarm mode however we do not have this worry. Instead, swarm adds a container to a *service*, meaning a user publishes ports on a service basis and not on a container basis as before. This in turn means that when a node receives a request on port 80 where 2 nginx containers are running the swarm can internally route traffic to any container which is a member of the nginx service. An illustration of this can be seen in Figure 12.

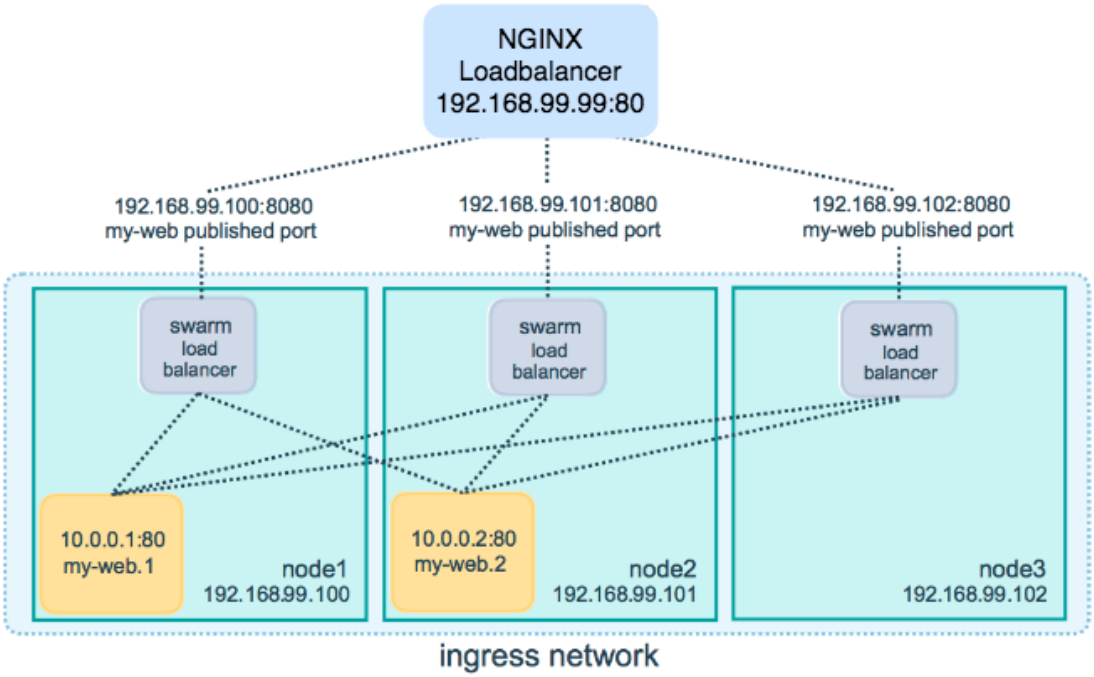

Figure 12: *Swarm Loadbalancer*

5 Conclusion

Since this paper dealt with two separate technologies in equal measure, Docker Swarm and Ansible, conclusions can also be drawn individually.

5.1 Ansible

Ansible is an extremely powerful tool. It is designed to allow for remote management of servers but this does not accurately describe the vast array of possibilities it carries in its tool belt. With one simple playbook and some related Ansible roles (Appendix A) we can instantly bring up a complete application environment. The aim of this report was to show how the automation of cloud infrastructure paired with application deployment can decrease the overhead in operations. I feel that this paper has proved this, as Ansible has shown to be extremely versatile, allowing the user to manage a remote system with no more difficulty than a local one.

In addition to deployment of an application - Ansible could be used as an 'application manager' also. Some examples of use cases of Ansible here would be, as discussed in Section 4.6, triggering updates of instances applications are hosted on. This could mean applying security patches, adding new software components, or provisioning further infrastructure.

5.2 Docker Swarm

We have also seen how Docker Swarm can provide an environment in which to reliably run Docker containers. In comparison with other solutions such as Kubernetes and Amazon ECS, Docker Swarm offers a native solution with no extra configuration on top of the regular Docker installation. It provides a mechanism for developers to run a completely containerised application without concerning themselves about how these containers are managed, where *exactly* they are being run from, and how they interact with each other. Effectively, swarm allows developers to interact with a collections of Docker hosts as if they were one single host.

However, Swarm also has some drawbacks. Swarm itself has been part of Docker for several releases, however 'Swarm mode' is only a recent development (version 1.12). This means it is in its early stages and lacks some important features. For instance it does not currently support automatic scaling based on common statistics such as CPU utilisation and network activity.

5.3 Further Investigation

Some features which were not included in this report but which I would have liked to look at were autoscaling applications running in a swarm. One possible solution here would be to use AWS autoscaling to create new instances based on set rules. Once these instances are created we could (through scripting or otherwise) add them to the Docker Swarm. Once the instance is added to the swarm it will then run services as required. However this could cause service interruptions if not configured correctly since Amazon's auto scaling would have no knowledge of Docker swarm. If it scaled down too quickly and the scale of a swarm service was not high enough it could mean (if only for an instance) that no containers of a certain application were running.

5.4 Closing Statement

Overall Docker Swarm, while a relatively new product, is certainly an excellent offering from Docker. It allows developers to use containers in their applications but treat them as regular applications.

Swarm paired with a modern provisioning tool such as Ansible allows for a powerful application environment to be created with ease. Also, because of the simplicity of both the learning curve is low which means developers do not need to invest large amounts of time to use these tools. While Ansible is well established Docker Swarm still has some way to go.

Appendices

A Complete Ansible Repository

https://github.com/StephenCoady/Docker-Swarm-With-Ansible

Bibliography

Ansible (2016), 'Ansible', https://www.ansible.com. Accessed: 15-11-2016.

DataDog (2015), 'Docker adoption', https://www.datadoghq.com/docker-adoption/. Accessed: 03-10-2016.

Docker (2016*a*), 'Docker Engine'.
URL: *https://www.docker.com/products/docker-engine*

Docker (2016*b*), 'Docker Hub', https://hub.docker.com/. Accessed: 03-10-2016.

Docker (2016*c*), 'Docker Swarm', https://docs.docker.com/swarm/overview/. Accessed: 03-10-2016.

Docker (2016*d*), *Swarm Mode Overview*. Accessed: 27-11-2016.
URL: *https://docs.docker.com/engine/swarm/*

Docker (2016*e*), 'What is docker?', https://www.docker.com/what-docker. Accessed: 03-10-2016.

Hochstein, L. (2013), *Ansible Up & Running*, O'Reilly.

Hykes, S. (2014), 'Docker 0.9: Introducing Execution Drivers And Libcontainer', https://goo.gl/07QUi7. Accessed: 03-10-2016.

Kerrisk, M. (2013), 'Namespaces in operation, part 1: namespaces overview [LWN.net]', https://lwn.net/Articles/531114/. Accessed: 03-10-2016.

Kubernetes (2016), 'Kubernetes', http://kubernetes.io/docs/whatisk8s/. Accessed: 15-11-2016.

Logan, M. (2014), 'Devops.com', https://devops.com/2014/01/23/fresh-stats-comparing-traditional-it-and-devops-oriented-productivity/. Accessed: 03-10-2016.

LXC (2016), 'Linux Containers', https://linuxcontainers.org/. Accessed: 15-11-2016.

Marks, M. (2016), *Swarm Mode Visualiser*. Accessed: 27-11-2016.
URL: *https://github.com/ManoMarks/docker-swarm-visualizer*

Matthias, K. and Kane, S. (2015), *Docker: Up & Running*, O'Reilly.
URL: *https://goo.gl/YqyFMj*

Mueller, E., Wickett, J., Gaekwad, K. and Karayanev, P. (2016), 'What is devops', https://theagileadmin.com/what-is-devops/. Accessed: 03-10-2016.

The Boto Project (2016), *Boto*. Accessed: 15-11-2016.
URL: *https://boto3.readthedocs.io/en/latest/*