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 22, 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.

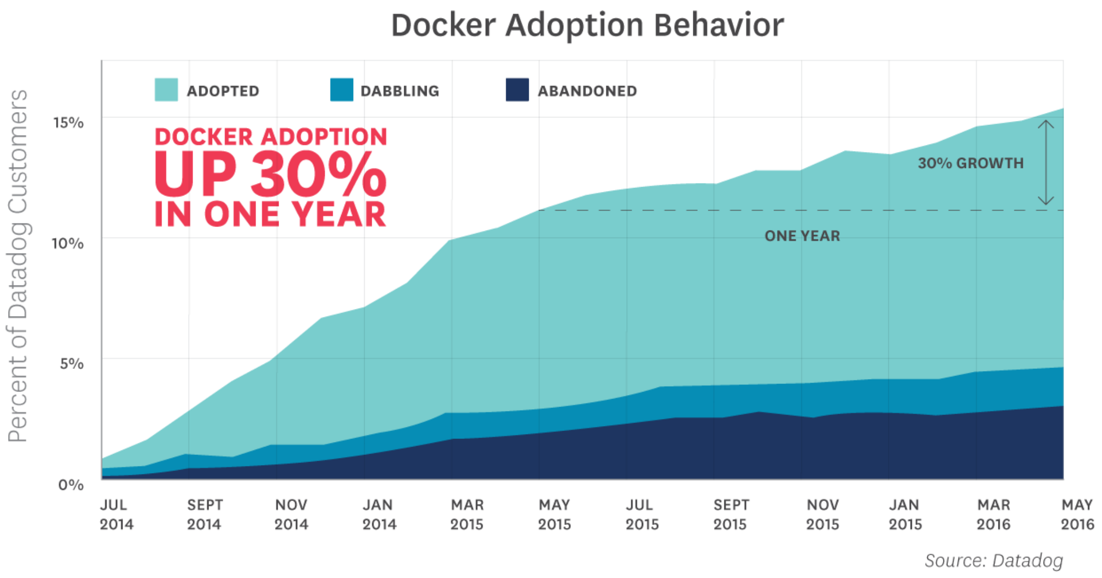

Figure 1: *Rise in Docker Usage. Credit: (DataDog, 2015)*

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*d*). 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.

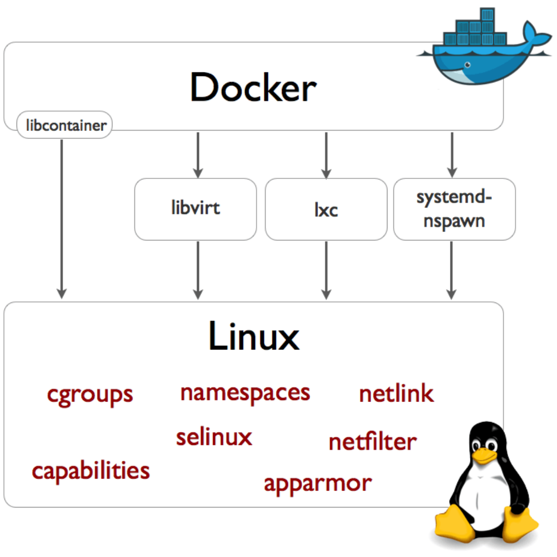

Figure 2: *Docker Using libcontainer. Credit: (Hykes, 2014)*

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 begins 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"
---
```

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 below 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.

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
---
```

3.3.4 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:

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

All of these steps will then be automated using Ansible. The final infrastructure model can be seen in Figure 3.

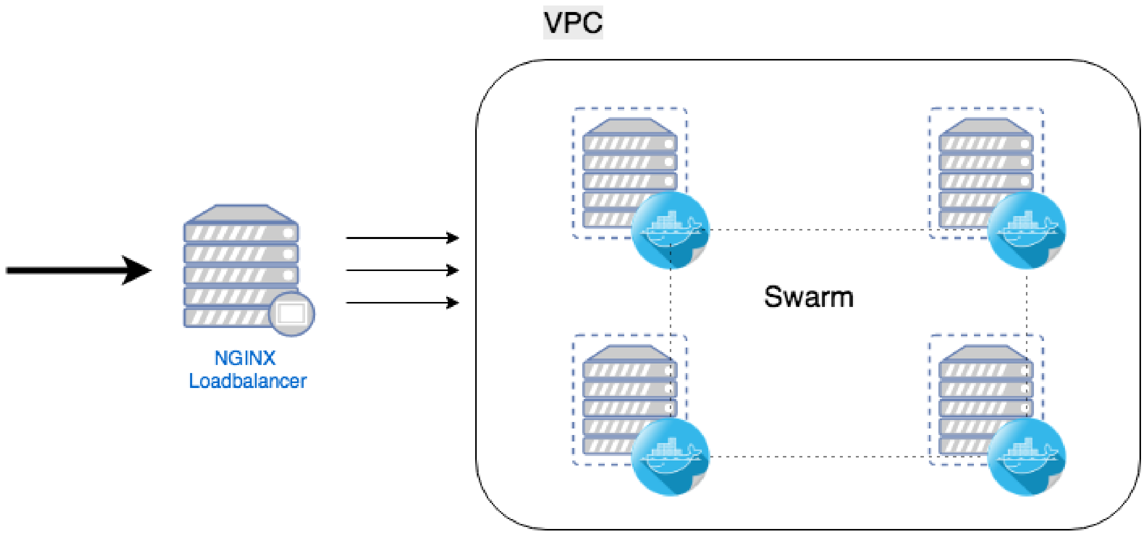

Figure 3: *Final Application Setup*

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:

Listing 3: create-instances.yml

```yaml
---
# 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
```

- Management
- Node
- Loadbalancer

We will discuss what these terms mean further in section 4.3, 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 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}}","Environment":"{{e
     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

Listing 5: install-docker-engine.yml

```
---
# 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
---
```

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 below in Listing 5.

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

Listing 6: install-docker-engine.yml

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

4.3 Creating the Swarm

5 Conclusion

6 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*), '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.

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/*