



Waterford Institute *of* Technology

CLOUD COMPUTING

BACHELOR OF SCIENCE (HONS) APPLIED COMPUTING

---

# Docker & ECS

---

Ciaran ROCHE - 20037160

April 16, 2019

## **Plagiarism Declaration**

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	What is a Container . . . . .	5
2.2	Control Groups . . . . .	6
2.3	Namespaces . . . . .	6
2.4	Capabilities . . . . .	8
2.5	Docker . . . . .	8
<b>3</b>	<b>Docker</b>	<b>10</b>
3.1	Building the Images . . . . .	10
3.2	Running the Containers . . . . .	10
3.3	Pushing Images to Quay.io . . . . .	12
3.4	Dockerfile . . . . .	13
3.5	Docker Compose . . . . .	15
3.6	RabbitMQ GoLang Producer . . . . .	17
3.7	RabbitMQ GoLang Consumer . . . . .	17
3.8	Docker Image Improvements . . . . .	17
3.9	Docker Summary . . . . .	18
<b>4</b>	<b>Amazon ECS</b>	<b>19</b>
4.1	Prerequisite . . . . .	19
4.2	Creating an ECS cluster . . . . .	19
4.3	Creating an ECS Instance . . . . .	19
4.4	Deploying an Application . . . . .	21
4.5	ECS Introduction Summary . . . . .	23
<b>5</b>	<b>AWS Fargate</b>	<b>24</b>
5.1	Fargate Steps Taken . . . . .	24
5.2	Fargate Summary . . . . .	27
<b>6</b>	<b>Summary</b>	<b>28</b>
	<b>Appendices</b>	<b>30</b>
A	RabbitMQ GitHub Repo . . . . .	30
B	RabbitMQ Manager Quay.io Repo . . . . .	30
C	RabbitMQ Producer Quay.io Repo . . . . .	30
D	RabbitMQ Consumer Quay.io Repo . . . . .	30

E	RabbitMQ Consumer . . . . .	30
F	RabbitMQ Producer . . . . .	30
G	ECS Demo PHP Simple App . . . . .	30
<b>Bibliography</b>		<b>31</b>

# 1 Introduction

The purpose of this paper is to document work carried out during multiple labs as part of the Cloud Computing module. The aim of these labs was to gain experience and exposure to Container technologies. Docker was introduced with a practical element of running RabbitMQ in containers and storing the images created in a repository. This can be seen details in Section 3. ECS was the second technology explored, the finding and steps taken can be seen in Section 4. Included in this paper is a container theory section which has been taken directly from a paper completed for Network and System Security module. It was decided to include in this paper as it provides some greater understanding and context to the technologies explored in this paper.

## 2 Theory

The following sub sections of theory have been taken from a report completed for Network and System Security, to skip these sub sections go directly to Section 3 for all work complete as part of this report.

### 2.1 What is a Container

Containers are a modern version of capabilities that have been available in Unix operating system Solaris for decades, in that you can run processes in isolation to the rest of the operating system (*Oracle Solaris 11*, n.d.). When it comes to applications, traditionally one would have a virtual machine which would house the application. This would mean the applications processes would run in isolation to the operating system on its host but it does this with the cost of a massive over head in that of the entire virtual machines operating system consumers memory and resources (*Yegulalp*, 2018).

Containers are a means that looks at how to run applications and processes in isolation, and they do this by giving applications a partitioned segment of the hosts operating system. This is managed by the container run time environment, while there is many run time environments that support OCI containers for the majority of this paper the focus will be on Dockers run time called libcontainer (*What is a Container*, 2018).

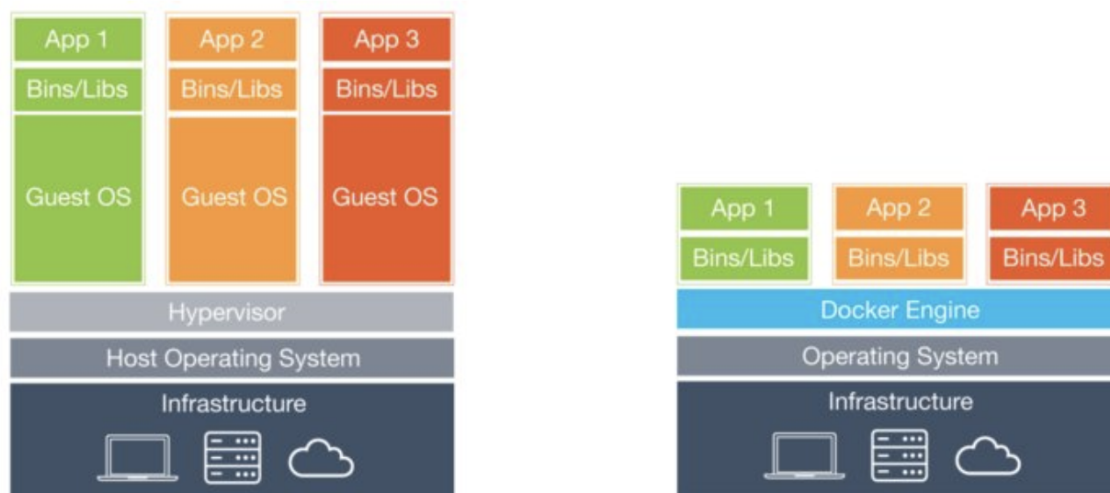


Figure 1: *Docker Container vs Virtual Machine (image from Docker)*

Figure 1 paints a clear picture of the difference between a conventional system of applications running

on a virtual machines (VMs) and applications running in containers. As can be seen from the figure, VMs are an abstraction of the physical hardware. The hypervisor allows multiple VM's run on a single hosts operating system. Each VM has its own operating system along with all the binaries, libraries and packages needed for the applications.

Where as in the Figure 1 it can be seen that containers are more of an abstraction at the application layer. As with VMs multiple containers can run on the same machine, but unlike VMs, containers share the OS kernel, all running in isolation based on a given space (*What is a Container*, 2018).

An understanding of cgroups and namespaces is needed to understand how containers share the OS kernel. It is these features that build the boundaries between containers and processes running on a host (Yegulalp, 2018).

## 2.2 Control Groups

Control Groups or Cgroups for short allow for the allocation of resources such as CPU time, system memory, network bandwidth or a combination of the resources. cgroups allow for the fine-grained control over allocating, prioritizing, denying and managing system resources. Management of these resources increase overall efficiency.

It must be noted that all processes on a Linux based system are child processes of a common parent. The *init* process, which is executed at boot time by the kernel starts all other processes. Thus the Linux process model is a single hierarchical tree.

Cgroups share similarities in that they are hierarchical and child cgroups inherit some attributes from their parent cgroup. This allows for simultaneous cgroups on a system. To paint a clearer picture if we describe the Linux process model as a single tree of processes, then the cgroup model is made up of one or many unconnected trees of processes (*Red Hat*, 2018).

## 2.3 Namespaces

Namespaces act similar to cgroups in that they deal with resource isolation, but unlike cgroups which deals with a number processes, Namespaces only isolate a single process. Linux Namespaces wrap a global system resource and makes it appear to the processes within its namespace that the global system resource is dedicated to that process (*Linux Manual*, n.d.a).

Furthermore these namespaces are broken into six different types, each namespace wraps a particular global resource. Due to this the overall nature of namespaces is to support the implementation of containers (Kerrisk, 2013).

### 2.3.1 UTS namespaces

In the space of containers, the UTS namespaces allow each container to have its own hostname and NIS domain name. This is used for the initialization and config scripts that take action based on the namespace. The term UTS gets its name from the 'Unix Time-Sharing System" (Kerrisk, 2013)

### 2.3.2 IPC namespaces

IPC namespaces allow for its own interprocess communication resource. In other words it provides shared memory spaces for accelerated communication (POSIX message queue filesystem) (Kerrisk, 2013).

### 2.3.3 PID namespaces

PID namespaces isolate the process ID number space. To put differently processes in different PID namespaces can have the same PID. The main benefit to this is that containers can be migrated between hosts and keep the same process IDs for all processes inside the container (Kerrisk, 2013).

### 2.3.4 Network namespaces

Network namespaces provide isolation network resources in that each network namespace has its own network devices, IP routing tables, port numbers, IP addresses etc. Network namespaces make containers extremely powerful from a network perspective. In that you could have multiple containers on the same host all bound to port 80 in their own network namespace (Kerrisk, 2013). Given that each container has its own virtual networking, tools like Docker Swarm and Kubernetes utilize this and provide suitable networking rules on the host.

### 2.3.5 User namespaces

User namespaces isolate the user and group ID number spaces. This means a process's user and group ID's can be different inside and outside of a user namespace. A typical use case would be a process would have full root privileges for operations within the user namespace but is unprivileged for operation outside the namespace (Kerrisk, 2013).



### 2.3.6 Mount namespaces

Mount namespaces isolate filesystem mount points seen by a group of processes. This allows for processes in different mount namespaces to have different views of a filesystem hierarchy (Kerrisk, 2013).

## 2.4 Capabilities

Traditional UNIX systems has two categories of processes, privileged and unprivileged. Privileged processes are identified as user ID 0, or also known as superuser or root. Where as unprivileged processes have a UID of nonzero (*Linux Manual*, n.d.b).

## 2.5 Docker

Dockers underlying architecture can be described as a client and server (docker daemon) based. This is shown in Figure 2, the server receives requests from the client through a RESTful API. The API along with a command line client are shipped by Docker. Due to make up of the architecture it allows for the docker daemon and client to run on the same machine or alternatively the client can connect to a remote docker daemon (Rad et al., 2018). It is the Docker daemon which orchestrates and manages all containers and images on the host where as the client is the control of the daemon.

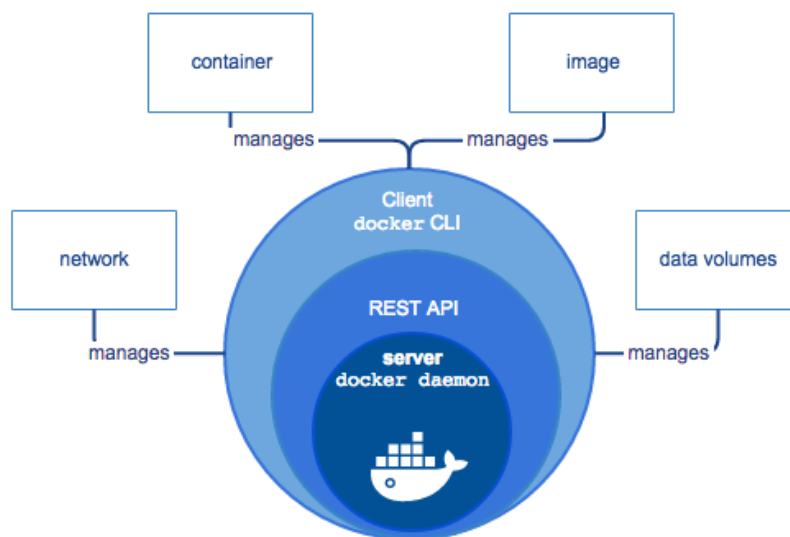


Figure 2: *Docker Architecture (image from Docker)*

Also depicted in Figure 2 is a Docker Image. A Docker image is essentially an immutable snapshot of a container. Running a Docker build command creates images. It is from this created image that a container is created by the Docker run command. Generally images are stored on a registry and will be demonstrated in Section 3.

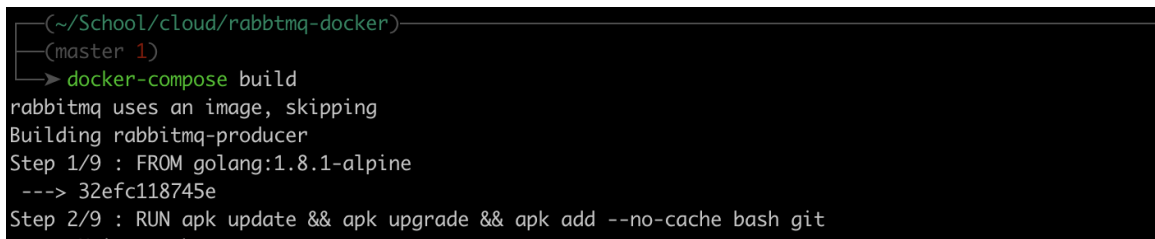
Docker Hub would be a main source for storing Docker images, and it itself is a huge security risk, as the Hub allows anybody to push their images to be used by themselves or other people. A study carried out in 2016 showed out of 352,416 community images tested on Docker Hub the worst image contained almost 1,800 vulnerabilities, with of 3,892 official images tested the worst image contained almost 800 vulnerabilities (Colyer, 2017).

## 3 Docker

The resulting images from the work below has been pushed to Quay.io, these link can be found at Appendices B, C and D. The application code including Dockerfiles and a Docker Compose file can be found on GitHub the URL is at Appendix A.

### 3.1 Building the Images

Docker Compose is a tool that is used for defining and running multiple Docker applications. We will look at the Docker Compose file in more detail in Section 3.5. For this Section we will focus on building the images. Once in our working directory running the command *docker-compose build* will build all our images for us following the details outlined in our Docker Compose File. If we look at Figure 3 we can see that the build steps followed are exactly as we would expect from a regular *docker build* command. This is because we have referenced the Dockerfile for each of our application in the Docker Compose file.

A terminal window with a dark background and light-colored text. The prompt is (~/.School/cloud/rabbitmq-docker) and the branch is (master 1). The command docker-compose build is entered. The output shows that rabbitmq uses an existing image and is skipped. Then, rabbitmq-producer is built. Step 1/9 is FROM golang:1.8.1-alpine, resulting in image 32efc118745e. Step 2/9 is RUN apk update && apk upgrade && apk add --no-cache bash git.

```
(~/School/cloud/rabbitmq-docker)
(master 1)
> docker-compose build
rabbitmq uses an image, skipping
Building rabbitmq-producer
Step 1/9 : FROM golang:1.8.1-alpine
--> 32efc118745e
Step 2/9 : RUN apk update && apk upgrade && apk add --no-cache bash git
```

Figure 3: *Docker Compose Build*

### 3.2 Running the Containers

With our images built we can create our containers. Just to note, an image is considered as a share-able snapshot of our application, where as a container is an image that is running.

Docker Compose has an *up* command which is the equivalent to executing *docker run* on each of our applications. This can be seen in Figure 4.

Figure 6 shows our running RabbitMQ management Dashboard where as Figure 5 shows the outputs from our producer sending messages and our consumer receiving messages.

```

(~/.School/cloud/rabbitmq-docker)
(master 1)
> docker-compose up
Starting rabbitmq-docker_rabbitmq_1 ... done
Starting rabbitmq-docker_rabbitmq-producer_1 ... done
Starting rabbitmq-docker_rabbitmq-consumer_1 ... done
Attaching to rabbitmq-docker_rabbitmq_1, rabbitmq-docker_rabbitmq-producer_1, rabbitmq-docker_rabbitmq-consumer_1
rabbitmq-producer_1 | Starting RabbitMQ producer...
rabbitmq-consumer_1 | Starting RabbitMQ consumer...
rabbitmq_1          |
rabbitmq_1          |
rabbitmq_1          |      ##  ##      RabbitMQ 3.6.9. Copyright (C) 2007-2016 Pivotal Software, Inc.
rabbitmq_1          |      ##  ##      Licensed under the MPL. See http://www.rabbitmq.com/

```

Figure 4: *Docker Compose UP*

```

rabbitmq-consumer_1 | 2018/11/06 12:03:34 [*] Waiting for messages. To exit press CTRL+C
rabbitmq-consumer_1 | 2018/11/06 12:03:34 Received a message: Hello RabbitMQ message 1
rabbitmq-producer_1 | 2018/11/06 12:03:39 [x] Sents Hello RabbitMQ message 2
rabbitmq-consumer_1 | 2018/11/06 12:03:39 Received a message: Hello RabbitMQ message 2
rabbitmq-producer_1 | 2018/11/06 12:03:44 [x] Sents Hello RabbitMQ message 3
rabbitmq-consumer_1 | 2018/11/06 12:03:44 Received a message: Hello RabbitMQ message 3
rabbitmq-producer_1 | 2018/11/06 12:03:49 [x] Sents Hello RabbitMQ message 4
rabbitmq-consumer_1 | 2018/11/06 12:03:49 Received a message: Hello RabbitMQ message 4
rabbitmq-producer_1 | 2018/11/06 12:03:54 [x] Sents Hello RabbitMQ message 5
rabbitmq-consumer_1 | 2018/11/06 12:03:54 Received a message: Hello RabbitMQ message 5

```

Figure 5: *RabbitMQ Producer and Consumer*

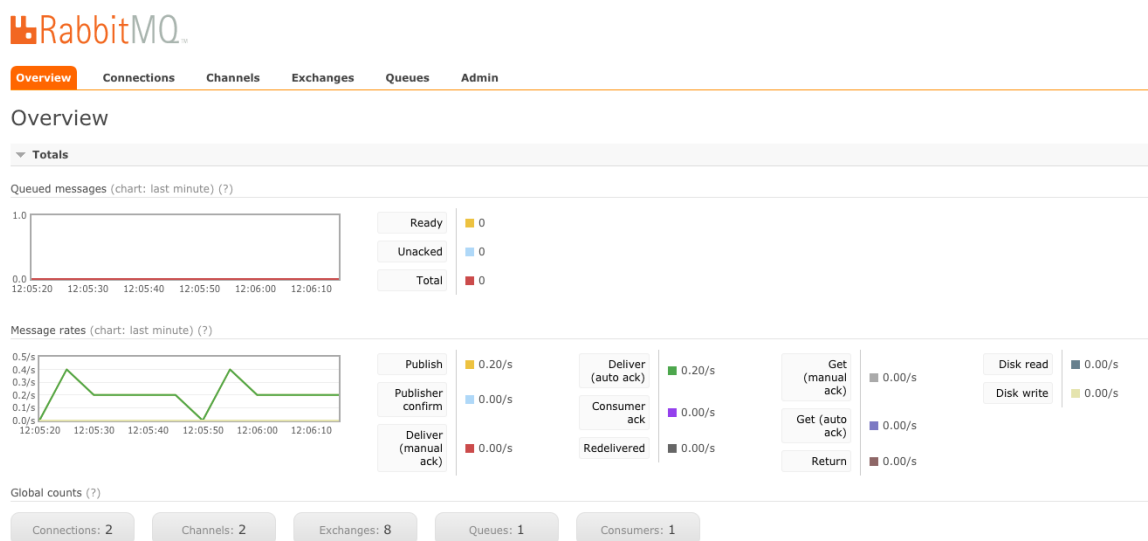
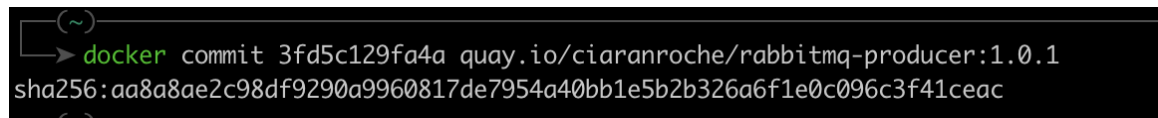


Figure 6: *RabbitMQ Dashboard*

### 3.3 Pushing Images to Quay.io

Now with our RabbitMQ Docker Application running we can safely commit our work and push it to a repository. Repositories like Docker Hub and Quay.io allow people to push images to store them and to have them referenced and pulled down at a later stage. While this is heavily based on trust as anyone is free to push and pull images, it is worth knowing who owns the images you wish to use and how secure these images are. For this reason I like to use Quay.io over Docker Hub, while both have the same features I feel you get a lot more out of the box from Quay.io. For instance Quay.io has a free and trusted vulnerability scanner which scans all your images, this can be incorporated into your CI/CD work flow to insure image integrity.

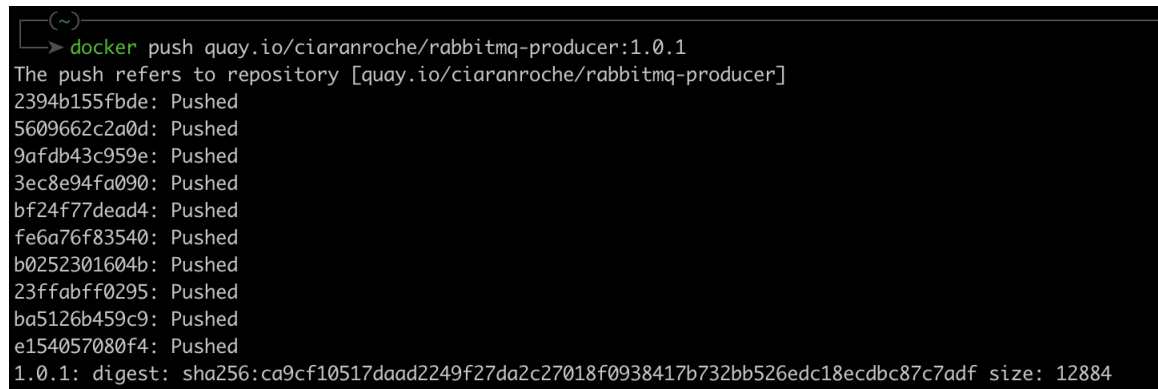
The first step we need to do over the conventional docker push to Docker Hub, we need to commit the image, this can be seen in Figure 7. We use `docker commit` followed by the id of our local image then we tag it starting with `quay.io/` followed by our Quay.io username followed by the image title and the version we wish to tag it as.

A terminal window with a dark background and light text. The prompt is a tilde symbol (~). The command entered is `docker commit 3fd5c129fa4a quay.io/ciaranroche/rabbitmq-producer:1.0.1`. The output is `sha256:aa8a8ae2c98df9290a9960817de7954a40bb1e5b2b326a6f1e0c096c3f41ceac`.

```
(~)
➔ docker commit 3fd5c129fa4a quay.io/ciaranroche/rabbitmq-producer:1.0.1
sha256:aa8a8ae2c98df9290a9960817de7954a40bb1e5b2b326a6f1e0c096c3f41ceac
```

Figure 7: *Docker Commit Producer*

Once we have committed our image we can push it as we would to Docker Hub. The Docker CLI picks up on the Quay.io tag and pushes the image to your repository. This can be seen in Figure 8. The steps mentioned were repeated for our 3 images and can be seen in Figures 9 and 10.

A terminal window with a dark background and light text. The prompt is a tilde symbol (~). The command entered is `docker push quay.io/ciaranroche/rabbitmq-producer:1.0.1`. The output shows the push refers to repository [quay.io/ciaranroche/rabbitmq-producer] and lists several layers being pushed: 2394b155fbde, 5609662c2a0d, 9afdb43c959e, 3ec8e94fa090, bf24f77dead4, fe6a76f83540, b0252301604b, 23ffabff0295, ba5126b459c9, e154057080f4. The final line shows the digest: `1.0.1: digest: sha256:ca9cf10517daad2249f27da2c27018f0938417b732bb526edc18ecdbc87c7adf size: 12884`.

```
(~)
➔ docker push quay.io/ciaranroche/rabbitmq-producer:1.0.1
The push refers to repository [quay.io/ciaranroche/rabbitmq-producer]
2394b155fbde: Pushed
5609662c2a0d: Pushed
9afdb43c959e: Pushed
3ec8e94fa090: Pushed
bf24f77dead4: Pushed
fe6a76f83540: Pushed
b0252301604b: Pushed
23ffabff0295: Pushed
ba5126b459c9: Pushed
e154057080f4: Pushed
1.0.1: digest: sha256:ca9cf10517daad2249f27da2c27018f0938417b732bb526edc18ecdbc87c7adf size: 12884
```

Figure 8: *Docker Push to Quay.io*

```
(~)
➤ docker commit b3183cd7a155 quay.io/ciaranroche/rabbitmq-consumer:1.0.1
sha256:9cd331871c634c57e4d59672a6a8aeec7f2a369cd6536a195ee4e1e5181c66aa
(~)
➤ docker push quay.io/ciaranroche/rabbitmq-consumer:1.0.1
The push refers to repository [quay.io/ciaranroche/rabbitmq-consumer]
04b8832a4c65: Pushed
5609662c2a0d: Mounted from ciaranroche/rabbitmq-producer
9afdb43c959e: Mounted from ciaranroche/rabbitmq-producer
3ec8e94fa090: Mounted from ciaranroche/rabbitmq-producer
bf24f77dead4: Mounted from ciaranroche/rabbitmq-producer
fe6a76f83540: Mounted from ciaranroche/rabbitmq-producer
b0252301604b: Mounted from ciaranroche/rabbitmq-producer
23ffabff0295: Mounted from ciaranroche/rabbitmq-producer
ba5126b459c9: Waiting
e154057080f4: Waiting
```

Figure 9: *Docker Commit and Push Consumer*

```
(~)
➤ docker commit 3fcbca478fb1 quay.io/ciaranroche/rabbitmq:3.6.9-management-alpine
sha256:d40d907fb42d20a2ae76b64dc9854588bfadebbb29244b8359fad1b701cdccac
(~)
➤ docker push quay.io/ciaranroche/rabbitmq:3.6.9-management-alpine
The push refers to repository [quay.io/ciaranroche/rabbitmq]
2e04d75ab543: Pushed
7d523ca99cda: Pushed
089493f661f9: Pushed
8795edbf76f6: Pushed
e4dd5e16cdf4: Pushed
72a7c61fd1bd: Pushed
c2e28b6d940d: Pushing [=====>] 3.443MB/5.672MB
f2575adcf1f4: Pushing [=====>] 7.913MB/27.82MB
f65eea9fd0f1: Pushed
1d2fec6366e1: Pushing [=====>] 12.29kB
e154057080f4: Waiting
```

Figure 10: *Docker Commit and Push Manager*

### 3.4 Dockerfile

Figure 11 shows our RabbitMQ Docker Application Structure. As we can see both our producer and consumer contain a Dockerfile. A docker file allows for the automation of instructions on building a Docker Image. These Commands could be entered manually but it is more efficient to package them in a Docker file and execute the *docker build* command to run each instruction. Starting from the top of the file and working down.

Figure 12 shows the dockerfile from our consumer application. It is identical to that of our producer only differences is where we reference the word *consumer* it is replaced with *producer*.

```
(~/School/cloud/rabbitmq-docker)
(master 1)
➤ tree -L 2
.
├── consumer
│   ├── Dockerfile
│   └── consumer.go
├── docker-compose.yml
└── producer
    ├── Dockerfile
    └── producer.go

2 directories, 5 files
```

Figure 11: *RabbitMQ Docker App Folder Structure*

```
1 FROM golang:1.8.1-alpine
2
3 RUN apk update && apk upgrade && apk add --no-cache bash git
4
5 RUN go get github.com/streadway/amqp
6
7 ENV SOURCES /go/src/github.com/ciaranRoche/rabbitmq-docker/
8 COPY . ${SOURCES}
9
10 RUN cd ${SOURCES}consumer/ && CGO_ENABLED=0 go build
11
12 ENV BROKER_ADDR amqp://guest:guest@localhost:5672/
13
14 WORKDIR ${SOURCES}consumer/
15 CMD ${SOURCES}consumer/consumer
```

Figure 12: *Dockerfile*

The instructions in the dockerfile can be broken down as follows:

- **FROM** First we need a base image, this is the image which our application will run on. As these are written in golang it was decided to go with a golang alpine image. Alpine images are a minimal Linux distro, it has a much smaller footprint in comparison to the likes of a Ubuntu or Fedora image. The golang alpine image contains dependencies needed to run application

written in golang.

- **RUN** Next we run an update and upgrade on our current alpine image, finally we add bash and git to our image. the `-no-cache` flag allows us to install packages that are not cached locally, thus keeping our footprint small.
- **RUN** We need to install a golang import that is used in our code, we do this using the `RUN` command where we execute a `go get` on our import. This will install our import in our gopath on the container.
- **ENV** We are now setting a environment variable titled `SOURCES` as we will reference this variable throughout the remaining steps. This variable is a path to our application in the container, and as can be seen from the figure we are following typical golang conventions.
- **COPY** Now we are copying all our files from our local directory to our application directory in the container.
- **RUN** Now we want to run a go build on our application, so we `cd` to our consumer directory and execute a `go build` command. We have set `CGO_ENABLED=0` to disable CGO, this will speed up subsequent builds as it will not rebuild packages that require C code to be compiled it will use the already built files.
- **ENV** We need to set another environment variable this is setting the ampq address. This is used to allow our application connect to RabbitMQ.
- **WORKDIR** We need to set our working directory to our application directory.
- **CMD** Finally we execute our built binary of our application.

### 3.5 Docker Compose

For the docker compose file it is broken into services and network. The network is defined and can be seen in Figure 13. It is given an arbitrary name, in this case I called it `sky-net` and set the driver to be a bridge. This will allow the containers to communicate via a bridge.

```
networks:
  sky-net:
    driver: bridge
```

Figure 13: *Docker Compose Network*

The services are broken down into the manager, producer and consumer. Figure 14 shows our rabbitmq manager service. We define the image to be used, in this case I updated it to match my own managment alpine which is pushed to Quay.io. I exposed the recommend ports and mapped



them to my localhost, configured environment variables for the log in. This is not recommend practise for production but for this scenario it will suffice. Finally we set the network. These steps could be entered in a single dirty one liner to run the RabbitMQ manager. As we can see using Docker Compose cleans the process and makes it easier to read.

```
services:
  rabbitmq:
    image: quay.io/ciaranroche/rabbitmq:3.6.9-management-alpine
    ports:
      - "4369:4369"
      - "5671:5671"
      - "5672:5672"
      - "15671:15671"
      - "15672:15672"
      - "25672:25672"
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    networks:
      - sky-net
```

Figure 14: *Docker Compose Manager*

Figure 15 shows how we define our Producer in the Docker Compose file. This is identical to how the Consumer is defined, the consumer definition is left out for brevity but can be seen in Appendix A.

We define some build steps, as Dockerfiles are included in the project we can simply point the docker compose to these files. Next we tag the image name. Set some environment variables to be used when running the container. We set a depends\_on to our manager, so that we ensure this container does not execute before the manager has. We set a link to the manager and finally define the network.

As before this is a much cleaner option to handling the lifecycle of a container.

```
rabbitmq-producer:
  build:
    context: .
    dockerfile: producer/Dockerfile
  image: rabbitmq-producer:1.0.1
  environment:
    - BROKER_ADDR=amqp://guest:guest@rabbitmq:5672/
    - QUEUE=test-queue
  depends_on:
    - rabbitmq
  links:
    - rabbitmq
  networks:
    - sky-net
```

Figure 15: *Docker Compose Producer*

### 3.6 RabbitMQ GoLang Producer

The code for the RabbitMQ GoLang Producer can be found in the Appendix F. The code logic is pretty simple in that it connects to RabbitMQ, and opens a channel. A queue is declared and then loop begins to send a message every 5 seconds.

### 3.7 RabbitMQ GoLang Consumer

The code for the RabbitMQ GoLang Consumer can be found at in the Appendix E. The code logic is pretty simple as with the Producer, a connection to RabbitMQ is made, a channel is opened, a queue declared and then the consumer is registered. It then continues to listen for messages.

### 3.8 Docker Image Improvements

While preparing for a presentation for a separate module I discovered a technique for building extremely efficient and secure GoLang Images.

The GoLang *build* command produces an executable binary. This means we can create an extremely lightweight image consisting of only our executable binary. Figure 16 shows our updated Dockerfile. We start by creating our image as before. This time we set the base image as *builder*. Once we run the GoLang build command and have got our executable binary in our builder image we begin building a new image. This image base will be *scratch*, this means we have a blank image, we simply add our envvar and then copy our executable from our builder image to our scratch image, set the entrypoint to the executable.

Figures 17 and 18 shows the before and after in the size of our images. It shows that we have reduced our consumer from 289MB to 4.57MB. The same steps were repeated for the producer and the same reduction was made.

```
FROM golang:1.8.1-alpine as builder

RUN apk update && apk upgrade && apk add --no-cache bash git

RUN go get github.com/streadway/amqp

ENV SOURCES /go/src/github.com/ciaranRoche/rabbitmq-docker/
COPY . ${SOURCES}

RUN cd ${SOURCES}consumer/ && CGO_ENABLED=0 go build

FROM scratch
ENV BROKER_ADDR amqp://guest:guest@localhost:5672/
COPY --from=builder /go/src/github.com/ciaranRoche/rabbitmq-docker/consumer/consumer /go/src/github.com/ciaranRoche/rabbitmq-docker/consumer/consumer

ENTRYPOINT [ "/go/src/github.com/ciaranRoche/rabbitmq-docker/consumer/consumer" ]
```

Figure 16: *Updated Docker File*

```
(~/School/cloud/rabbitmq-docker) (master 2)
> docker images | grep rabbitmq-consumer
rabbitmq-consumer 1.0.1 6ab0eb5
1f665 About a minute ago 289MB
quay.io/ciaranroche/rabbitmq-consumer 1.0.1 9cd3318
71c63 2 days ago 289MB
```

Figure 17: *Before Docker Image Size*

```
(~/School/cloud/rabbitmq-docker) (master 3)
> docker images | grep rabbitmq-consumer
rabbitmq-consumer 1.0.1 73d81ce
0d3f7 28 seconds ago 4.57MB
quay.io/ciaranroche/rabbitmq-consumer 1.0.1 9cd3318
71c63 2 days ago 289MB
```

Figure 18: *After Docker Image Size*

### 3.9 Docker Summary

This was the first look at Docker in an academic setting. Previous to this lab the exposure to Docker was from my own interest along with usage on work placements. Due to previous exposure I took the opportunity to practise what I learned and gain some more experience with the programming language GoLang mixed with adding the challenge of working with Docker Compose.

The buy in to Docker is quite low, in that you can Dockerize an application with minimal effort and have it running in isolation to all other applications on your hardware. But with experience Docker can become quite complex, especially when it comes to configuring the most efficient container possible for your applications.

From this paper and work completed in a separate Module, so far my findings is that Docker itself can only make an image so efficient, its the technologies you choose for your application that has the overall effect on image efficiency. This can be seen in Section 3.8, through utilizing GoLang's build a minimal image could be built.

## 4 Amazon ECS

This section was an introduction to Amazons Elastic Container Service. This is Amazons implementation of Container Orchestration. The work was carried out over two weeks. With the first week being an introduction to the environment and finally the second week has us using AWS Fargate to scale and load balance containers within a Serverless environment.

### 4.1 Prerequisite

The AWS CLI tool was need to complete this portion of the practical. With that an Access Key ID and Secret Access Key was added to the AWS configuration to allow the CLI communicate to AWS. The Docker Client and Docker Daemon was also need for the completion of this practical.

### 4.2 Creating an ECS cluster

This step was optional in a normal environment in that by standard when launching a container instance a default cluster is launched, this saves having to specify *cluster name* on commands. Due to the lab layout with multiple students using AWS Alias accounts individual clusters needed to be set up. The setting up of the cluster can be seen in Figure 19.



```
(~/School/cloud/rabbitmq-docker)
(master)
> aws ecs create-cluster --cluster-name CRClusterdemo
{
  "cluster": {
    "status": "ACTIVE",
    "statistics": [],
    "clusterName": "CRClusterdemo",
    "registeredContainerInstancesCount": 0,
    "pendingTasksCount": 0,
    "runningTasksCount": 0,
    "activeServicesCount": 0,
    "clusterArn": "arn:aws:ecs:eu-west-1:828000029458:cluster/CRClusterdemo"
  }
}
```

Figure 19: *Creating ECS Cluster*

### 4.3 Creating an ECS Instance

An ECS Instance is needed before any containers can be run. Figure 20 shows the AMI needed. This is a community edition ECS optimized Linux AMI.

Figure 21 shows the instance size that was selected, demo app including a Linux busy box and a PHP application it was deemed to use a t2.small, as it includes an extra GB of ram to in comparison to the t2.mini free tier compliant instance.

An IAM role was added and can be seen in Figure 22.

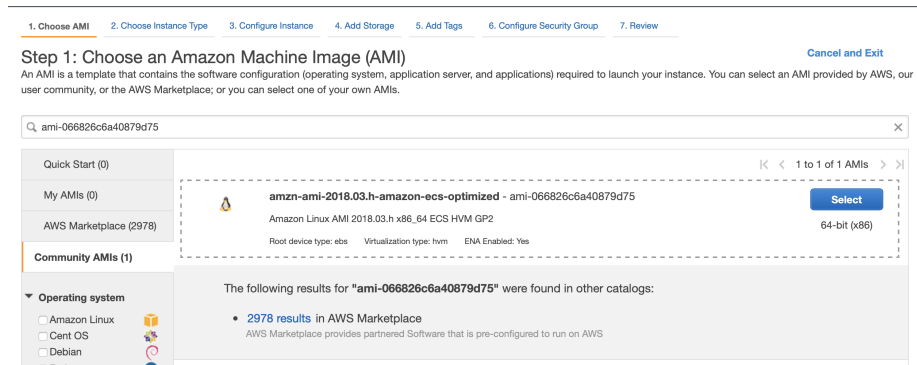


Figure 20: *Choosing ECS AMI*

<input checked="" type="checkbox"/>	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
-------------------------------------	-----------------	----------	---	---	----------	---	-----------------	-----

Figure 21: *Selecting Instance Size*



Figure 22: *Selecting IAM Role*

A bash script was added to the user details to allow the instance to be deployed to the cluster that was created. This can be seen in Figure 23



Figure 23: *Adding User Details*

With the instance launch the command seen in Figure 23 was used to ensure the instance was launched to the correct cluster.

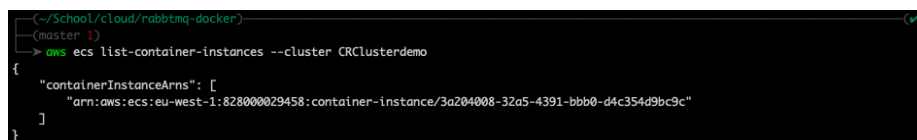
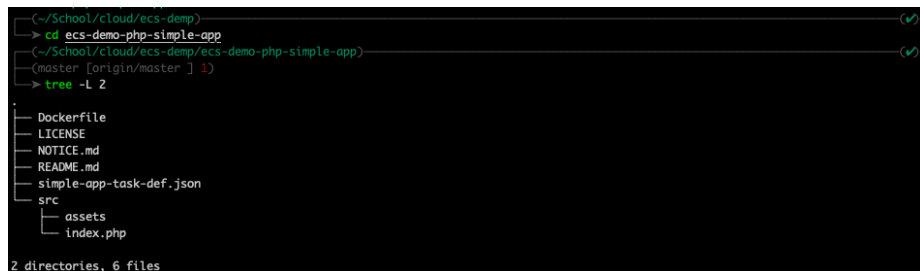


Figure 24: *Returning ECS Instances*

## 4.4 Deploying an Application

With our ECS instance successfully launched to our cluster the next step was to build our application image and run it within the ECS instance. The application used was the ecs demo application a link to the app repo can be found at Appendix G. Figure 25 shows the structure of the application. Two files are worth mentioning here:

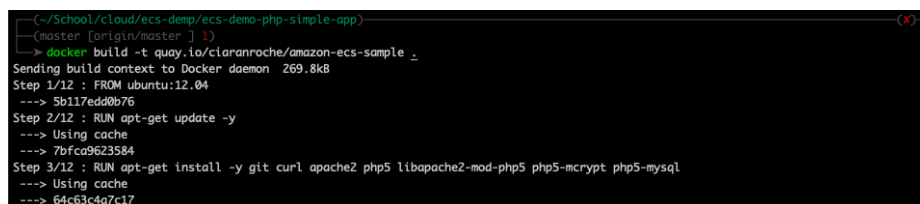
- **Dockerfile** - This is a regular dockerfile which is used to construct an image of our application
- **simple-app-task-def** - This file is specific to ECS in that it is used in the Registering of the Application in our ECS instance. This file is a way to automate the construction of our container in that the image is specified along with information like exposed ports, memory allocated, entrypoints etc.



```
(~/School/cloud/ecs-demp)
└─ cd ecs-demo-php-simple-app
   (~/School/cloud/ecs-demp/ecs-demo-php-simple-app)
   └─ (master [origin/master] 1)
      └─ tree -L 2
         .
         ├── Dockerfile
         ├── LICENSE
         ├── NOTICE.md
         ├── README.md
         ├── simple-app-task-def.json
         └── src
             ├── assets
             └── index.php
         2 directories, 6 files
```

Figure 25: *ECS Demo App*

Figure 26 shows the image being built using *docker build*. It can be seen from the image that we using the Quay.io naming convention so that when the image is pushed it will be saved to my Quay.io registry. This can be seen in Figure 27



```
(~/School/cloud/ecs-demp/ecs-demo-php-simple-app)
└─ (master [origin/master] 1)
   └─ docker build -t quay.io/claranroche/amazon-ecs-sample .
      Sending build context to Docker daemon 269.8kB
      Step 1/12 : FROM ubuntu:12.04
      --> 5b117edd0b76
      Step 2/12 : RUN apt-get update -y
      --> Using cache
      --> 7bfca9623584
      Step 3/12 : RUN apt-get install -y git curl apache2 php5 libapache2-mod-php5 php5-mcrypt php5-mysql
      --> Using cache
      --> 64c63c4d7c17
```

Figure 26: *Building Demo App Image*

With the image pushed to Quay.io we needed to update the *simple-app-task-def.json* file to specify the location of our application image which we want to run within our ECS instance. This can be seen in 28.

```
(~/School/cloud/ecs-demp/ecs-demo-php-simple-app)
(master [origin/master ] 1)
➔ docker push quay.io/ciaranroche/amazon-ecs-sample
The push refers to repository [quay.io/ciaranroche/amazon-ecs-sample]
99774727e265: Pushed
28720bd56d02: Pushed
7a9ee1cd7660: Pushed
6f0c3e56d82e: Pushed
31686a61bd14: Pushed
3236d873f79b: Pushing [=====] 28.41MB
3efd1f7c01f6: Pushed
73b4683e66e8: Pushed
ee60293db08f: Pushed
9dc188d975fd: Pushed
58bcc73dcf40: Pushing [=====] 78.96MB/103.4MB
[]
```

Figure 27: *Pushing Image to Quay.io*

```
"containerDefinitions": [
  {
    "environment": [],
    "name": "simple-app",
    "image": "quay.io/ciaranroche/amazon-ecs-sample",
    "cpu": 10,
    "memory": 200,
```

Figure 28: *Adding Image to App Definition*

With the *simple-app-task-def.json* updated Figure 29 shows the command used to register this information with ECS.

```
(~/School/cloud/ecs-demp/ecs-demo-php-simple-app)
(master [origin/master ] 1)
➔ aws ecs register-task-definition --cli-input-json file://simple-app-task-def.json
{
  "taskDefinition": {
    "status": "ACTIVE",
    "family": "console-sample-app",
    "placementConstraints": [],
    "compatibilities": [
      "EC2"
    ],
    "volumes": [
      {
        "host": {},
        "name": "my-vol"
      }
    ]
  }
}
```

Figure 29: *Registering App Definition*

Once the Definition was registered the ECS run task command was used to launch the container within our image this can be seen in Figure 30

```
(~/School/cloud/ecs-demp/ecs-demo-php-simple-app)
(master [origin/master ] 1)
> aws ecs run-task --cluster CRClusterdemo --task-definition console-sample-app:1 --count 1
{
  "failures": [],
  "tasks": [
    {
      "taskArn": "arn:aws:ecs:eu-west-1:828000029458:task/699a18c4-2f6e-4de0-9fd9-80b62ad83e47",
      "group": "family:console-sample-app",
      "attachments": [],
      "overrides": {
        "containerOverrides": [
          {
            "name": "simple-app"
          },
          {
            "name": "busybox"
          }
        ]
      }
    }
  ],
  "launchType": "EC2"
}
```

Figure 30: *Running App Instance in Cluster*

Figure 31 show that with the container launched we could access our PHP application via the ECS instance public IPV4 address

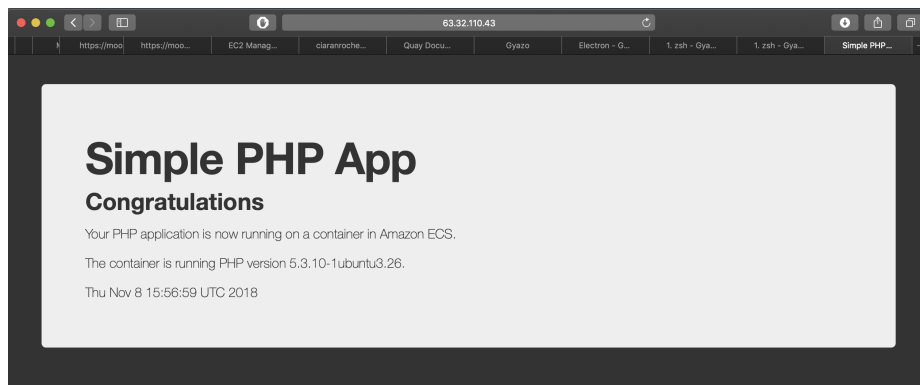


Figure 31: *Resulting Running App*

## 4.5 ECS Introduction Summary

From this element of the practical, as an introduction to ECS I am finding it hard to see the value in ECS as an infrastructure over using an EC2 instance with Docker. Looking more so at running multiple container I can see where the value may be added which will lead to the need for using AWS Fargate for the orchestration of the containers. Again fitting with the model of ending up vendor locked to AWS, my interest has been sparked at running an alternative to see the work involved in running a Kubernetes cluster on Amazon services be it on EC2 and EKS and comparing it to Fargate.



## 5 AWS Fargate

Following on from the introduction to ECS we looked at AWS Fargate. According to AWS, Fargate is a compute engine for ECS that abstracts the management of servers or clusters from the user who wants to run containers on AWS architecture (*AWS Fargate, 2018*). This practical element looks at the Fargate launch type, where we packaged up our apache website, specified some requirements, and launched the application. As ECS provides metrics out of the box we were able to add a scaling policy to our Application Load Balancer to automatically scale our containers.

### 5.1 Fargate Steps Taken

The first step was to create a cluster, for this practical I was using my own AWS account over the provided alias. This was partly due to my own experimentation with EKS and having to change over my AWS credentials. More on this later. Figure 32 shows the command used for the cluster creation. A cluster is a logical grouping of tasks and services. It is the cluster that provides the layer of abstraction away from the instances and regions. A task definition was needed to be registered.

```
~/School/cloud/ecs-dmp
$ aws ecs create-cluster --cluster-name CR-fargate-cluster
{
  "cluster": {
    "status": "ACTIVE",
    "statistics": [],
    "clusterName": "CR-fargate-cluster",
    "registeredContainerInstancesCount": 0,
    "pendingTasksCount": 0,
    "runningTasksCount": 0,
    "activeServicesCount": 0,
    "clusterArn": "arn:aws:ecs:eu-west-1:998447852043:cluster/CR-fargate-cluster"
  }
}
```

Figure 32: *Creating a Cluster*

This is required in order to run containers in ECS. In this task definition we are declaring our image as an Apache image. Along with passing some details such as cpu and memory usage limits. Figure 33 shows the command used to register the definition.

```
~/School/cloud/ecs-dmp
$ aws ecs register-task-definition --cli-input-json file://fargate-task.json
{
  "taskDefinition": {
    "status": "ACTIVE",
    "networkMode": "awsvpc",
    "family": "sample-fargate",
    "placementConstraints": [],
    "requiresAttributes": [
      {
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
      }
    ],
    "cpu": 256,
    "memory": 512
  }
}
```

Figure 33: *Registering a task definition*

A load balancer and a target group to be used within the ECS service needed to be created. Figure 34 shows the created Load Balancer. An application load balancer was used, this is to allow for dynamic mappings of ports used within the cluster.

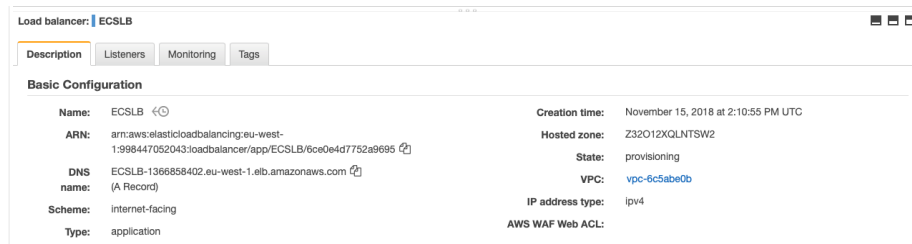


Figure 34: *Created load balancer*

Now with the task definition registered, the cluster created and a load balancer created we were ready to create our ECS service. Figure 35 shows my first attempt, in that an error was thrown. It showed that there was a role problem. As I was working off my own account I felt this was the problem. So I began troubleshooting it. A quick google search led to multiple people running into the same problem and all answers pointed to a IAM role problem. So I reconfigured my AWS CLI, this time using the not recommended steps, but configuring my root account CLI access. Thinking this would solve my problem. I killed all the previous steps and started again on my Root Account. Again I met this problem. Confused as my root account should not have any permission errors like this. I tried messing around with the command format, editing security groups, subnets etc. The error persisted.

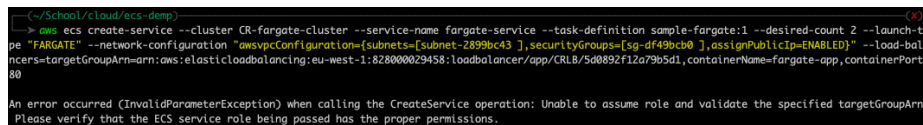


Figure 35: *ECS Service Creation Error*

Admitting defeat I reconfigured my AWS CLI to the alias account provided for this assignment, resigned into the alias account on the AWS console, and started the steps all over again. Once again reaching this problem, it occurred to me it was not an IAM role problem. My google search had led me down the wrong path. It was on my tenth read of the error did I notice it was failing at the targetGroupArn, target group being the key word, I was using the ARN for my load balancer and not the target group. Embarrassed by my error, I quickly changed the command to my target group, Figure 36 shows the successful Service creation.

```

~/School/cloud/ecs-demp
$ aws ecs create-service --cluster CR-fargate-cluster --service-name fargate-service --task-definition sample-fargate:1 --desired-count 2 --launch-type "FARGATE" --network-configuration "awsVpcConfiguration={subnets=[subnet-2899bc43 ],securityGroups=[sg-df49bcb0 ],assignPublicIp=ENABLED}" --load-balancers=targetGroupArn=arn:aws:elasticloadbalancing:eu-west-1:828000029458:targetgroup/CRTG/5333d896acc1c5df,containerName=fargate-app,containerPort=80

{
  "service": {
    "networkConfiguration": {
      "awsVpcConfiguration": {
        "subnets": [
          "subnet-2899bc43"
        ],
        "securityGroups": [
          "sg-df49bcb0"
        ],
        "assignPublicIp": "ENABLED"
      }
    }
  }
}

```

Figure 36: *ECS Service Creation Success*

Figure 37 shows we now have access to our Apache application via our Load Balancer DNS. Now



Figure 37: *Apache Application*

with the application running, it is time to update the task, as it would be nice to be able to manually test the Load Balancer is running as expected. So a new task definition was created and registered. Figure 38 shows the new tasks provisioning using the latest task definition. In this case due to using the allocated account the task definition is numbered 8.

Services Tasks ECS Instances Metrics Scheduled Tasks									
<div> <div>Run new Task</div> <div>Stop</div> <div>Stop All</div> </div> <div>Last updated on November 15, 2018 2:59:18 PM (0m ago)</div>									
<div> <div>Desired task status: Running Stopped</div> <div>Filter in this page</div> <div>Launch type: ALL</div> <div>&lt; 1-4 &gt; Page size 50</div> </div>									
<input type="checkbox"/>	Task	Task definition	Container inst...	Last status	Desired status	Started By	Group	Launch type	Platform versi...
<input type="checkbox"/>	506f01d0-3767...	sample-fargate:8	--	PENDING	RUNNING	ecs-svc/92233...	service:fargate-...	FARGATE	1.2.0
<input type="checkbox"/>	aa8ec124-2e22...	sample-fargate:1	--	RUNNING	RUNNING	ecs-svc/92233...	service:fargate-...	FARGATE	1.2.0
<input type="checkbox"/>	c5c6aeb2-231a...	sample-fargate:8	--	PENDING	RUNNING	ecs-svc/92233...	service:fargate-...	FARGATE	1.2.0
<input type="checkbox"/>	e96cfa9c-9b11...	sample-fargate:1	--	RUNNING	RUNNING	ecs-svc/92233...	service:fargate-...	FARGATE	1.2.0

Figure 38: *Updating Task Definition*

Figure 39 shows the new updated Apache application. As we can now see the hostname, subsequent refreshes show the new IP address. From visual inspection it appeared to act in a Round Robin fashion as it alternated between the containers. As mentioned AWS provide metrics allowing us to

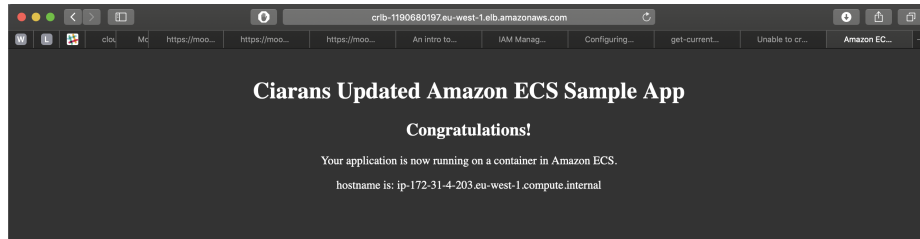


Figure 39: *Updated Application*

easily create scaling policies. So for this practical we created a policy to create a new task if the Load Balancer receives over 50 requests. Figure 40 shows the created scaling policy. A script was

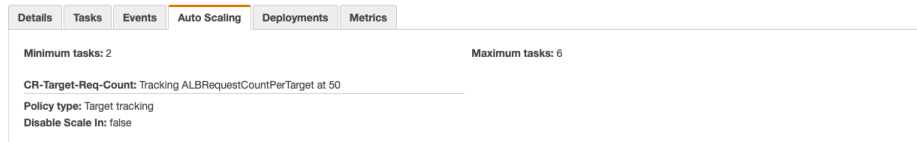


Figure 40: *Created Scaling Policy*

run to curl multiple requests on our Load Balancer, Figure 41 shows we now have 3 running tasks.

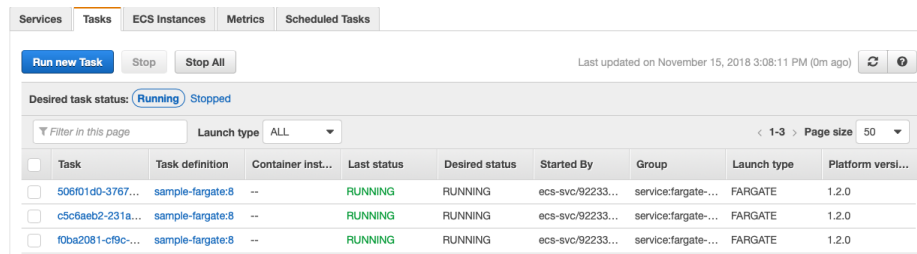


Figure 41: *Added Task*

## 5.2 Fargate Summary

Running into the problem I had creating the ECS service caused me to trawl over multiple pages of documents etc, which gave me an overall deeper insight into Fargate. It is clear the power Fargate bring to ECS creating a more rounded orchestration tool. I will go into more detail about this in Section 6.

## 6 Summary

As an introduction into Containers, this was an enjoyable number of labs. Taking Containers from running them locally to deploying them on AWS to using an orchestration tool. In this summary I will break down each section and my thoughts on them, albeit this may be an opinionated summary. RabbitMQ and Docker was enjoyable in that the majority of Docker tutorials revolve around some sort of Server creation, like express/node. Or implementing an instance of MongoDB for local development. While two great introductions, getting hands on with a Messaging system will be valuable going forward as development tends to lean closer to microservices over monoliths, the need for messaging is apparent. It also was nice to see a running application in that consuming and producing messages.

My main take away from this practical and work I completed in a separate module around Containers is that creating efficient containers is key. As mentioned the internet is rip with tutorials and blog post on how to dockerise an application. But if we tend to dive a little deeper and go beyond 'it just works' to creating an efficient container we not only reduce the size of the application, which is something that gets lost at this level as size is not important in that we are running one container locally, but if we are to scale a container across multiple instances size begins to become a factor quite quickly, but On the other hand, with efficiency increased we also make our container more secure. The smaller the container the smaller the attack surface.

This can be seen in Section 3.8, through how GoLang produces an executable binary, there was no need for a base image. I could build an image from scratch and just give it the binary. This greatly reduced the size and in turn increased the security while not affecting the logic of the application. So with that said, giving some extra time to think about the application and the technologies chosen it can pay off when it comes to container efficiency. Also not just accepting 'it works' and dive into how to actually configure an image and understand the steps and layers that make up an image is a tool required in any Software engineers toolbox today.

My final take away is the importance of orchestration tools, in that there is no need to provide a bash shell in a container, as we should not have to gain access to debug when a container is deployed. We should be debugging throughout development and use orchestration tools to roll out updates and patches.

This brings me nicely to the next section where we looked at ECS. ECS is Amazons orchestration service for Docker containers. A side not before continuing, while marketed as a service for Docker containers, Docker is OCI compliant meaning any container that is OCI compliant is able to run on ECS, these include tools like Buildah and Kata Containers. As these are compliant they are interoperable with Docker runtime.

Throughout the introduction to ECS I failed to see the value of it as a service. As there was some upskilling involved, as with any new technology or tool, but I felt the pay off was not worth it. I may be bias but the more I work with AWS the more I feel that while it is easy to get up and running

with their services for the most part, to really feel the benefit from them you have to buy into other services. Which always run the risk of becoming vendor locked. I felt ECS followed this pattern, that between the upskilling needed and the end result being a container running in AWS it didn't add any value over hosting a container on a regular EC2 instance.

This is where our third part of the assignment comes into play, it was an introduction to Fargate. Fargate gives us no clusters to manage per say and the ability to scale, all this interrogated into ECS. While this increase the value and functionality of ECS, I failed to be sold on the concept. From what I seen the following list is a number of observations I did not like:

- Abstraction, I as a lot of developers, do not like to much abstraction. ECS and Fargate provides too much abstraction to the underlying components and force particular patterns. With this abstraction in mind it leads me to my next point.
- Security, abstraction adds layers, and with added layers the attack surface of your deployment is increased, in this your deployment may be vulnerable. The extra layers may lead to overlooked security concerns. From spending some time in the documentation I found a lack of information on security practices which was worrying.
- Configuration, I found the overall setup to be burdened with steps and overly long one liners. A lot of the setup could be consolidated into simple config files, while supported I felt the documentation pushed more towards the one liners.
- Scaling Statefull Applications, From my investigation in order to perform scaling of statefull applications require another amazon product, EBS. As previously mentioned in order to get full benefit requires a bigger AWS stack
- Cost, as there is not a clear cut price in comparison to EC2 in that you know exactly how much you are being charged by the hour. Fargate comes at a cost of the cpu usage and memory usage per hour. This in my opinion is a downfall in that it may be hard to predict overall cost.

From getting some hands on with ECS and Fargate I decided to have a look at EKS, Amazons Kubernetes service. Out of the box Kubernetes provide all the functionality of ECS and Fargate, so I was intrigued to how AWS implemented it. While I found the setup to be slightly more complicated once set up I was able to play around with Kubernetes as I would with MiniKube through the kubectl. For me, with some Kubernetes experience I felt this was a much better approach as once I created my multi-node cluster there was no additional upskilling needed as I used tools I was already familiar with.

For me this was an enjoyable paper, and my main take away regardless of opinion on AWS, I felt the importance of Containers was brought across along with the value of being able to orchestrate multiple Containers.

## Appendices

### A RabbitMQ GitHub Repo

<https://github.com/ciaranRoche/rabbitmq-docker>

### B RabbitMQ Manager Quay.io Repo

<https://quay.io/repository/ciaranroche/rabbitmq>

### C RabbitMQ Producer Quay.io Repo

<https://quay.io/repository/ciaranroche/rabbitmq-producer>

### D RabbitMQ Consumer Quay.io Repo

<https://quay.io/repository/ciaranroche/rabbitmq-consumer>

### E RabbitMQ Consumer

<https://github.com/ciaranRoche/rabbitmq-docker/blob/master/consumer/consumer.go>

### F RabbitMQ Producer

<https://github.com/ciaranRoche/rabbitmq-docker/blob/master/producer/producer.go>

### G ECS Demo PHP Simple App

<https://github.com/aws-samples/ecs-demo-php-simple-app>

## Bibliography

*AWS Fargate* (2018).

**URL:** <https://aws.amazon.com/fargate/>

Colyer, A. (2017), ‘A study of security vulnerabilities on docker hub’.

**URL:** <https://blog.acolyer.org/2017/04/03/a-study-of-security-vulnerabilities-on-docker-hub/>

Kerrisk, M. (2013), ‘Namespaces in operation, part 1: namespaces overview’.

**URL:** <https://lwn.net/Articles/531114/>

*Linux Manual* (n.d.a).

**URL:** <http://man7.org/linux/man-pages/man7/namespaces.7.html>

*Linux Manual* (n.d.b).

**URL:** <http://man7.org/linux/man-pages/man7/capabilities.7.html>

*Oracle Solaris 11* (n.d.).

**URL:** <https://www.oracle.com/solaris/solaris11/>

Rad, B. B., Bhatti, H. J. and Ahmadi, M. (2018), ‘An introduction to docker and analysis of its performance’.

*Red Hat* (2018).

**URL:** [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/6/html/resource\\_management\\_guide/](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/)

*What is a Container* (2018).

**URL:** <https://www.docker.com/resources/what-container>

Yegulalp, S. (2018), ‘What is docker? docker containers explained’.

**URL:** <https://www.infoworld.com/article/3204171/docker/what-is-docker-docker-containers-explained.html>