NWEN 243 – 2024T2

Project 3 – Part A (of 2 parts): Making it Scale - Containers and Serverless Compute

Note: Part **B** is new in 2024 and needs another round of testing and familiarisation time for the tutors. As such it will be published one week after part **A**.

Due: See the ***submission system*** for authoritative dates and times, however ***both*** parts **A** and **B** will be submitted together.

FOR SUBMISSION:

- In terms of supporting evidence, it is up to you to take appropriate **screenshots** and make commentary.
  - You must put together a folio of evidence supporting your work in this lab.
  - In your screenshots ensure any IP addresses and/or instance names are visible.
  - Collect your screen shots and organise into a PDF with a narrative explaining what is going on, what each screen shot reveals and how they line up to the sections in this Project.
  - Label this narrative, **Project3partA.pdf**
- You've seen what we're after, by example from Project 2, you need to provide a similar level of evidence here and this is reflected in the marking criteria. Handing in some working code and the pdf equivalent of crumpled paper won't get you the marks you'd like - there needs to be evidence of introspection and understanding.
- There are also a couple of questions to answer at the end - you could make some tests or experiments and in general provide an insightful commentary of what is going on.

TODO

- As we saw in project 2, if we want to scale up using VMs to potentially millions of users we'd need many EC2 instances, and we'd need to create, launch, coordinate and load balance them. That's a lot of work. Alternatively,
- What if we want to keep our service small and boutique? In this case keeping even a small micro instances running is a waste of resources, perhaps even getting rid of the whole server idea and going serverless?

What we need to do is "right-size" our resources and the scale of our program to match our service - what we're doing in this project is to build a container for your Magic8Ball server, so that we can create any number of essentially identical duplicate containers and then look at scaling this out.

GETTING STARTED WITH CONTAINERS

There are lots of different ways to start - we can use a standard test Docker image, we could make our own image using docker, or other containerising service. Once we have created the image we need to load it into one of many possible choices of Docker registry, such as Amazon ECR, Artifactory, or Docker's own Registry). There are many paths to success here:

- My plan for this project is to do things as manually as possible, not because this is how its usually done, but because this is a learning exercise and much of the work on the cloud may as well be magic if you don't see what is going on 'under the hood'.
- Where we can - I will keep choices within the AWS ecosystem (plus Docker). I'm not suggesting that this is the best way to do things every time, indeed it is not - this is Docker unplugged! - but for learning the basics, sure why not!

Our first goal will be to Containerise your Magic8Ball server.

PLAN

Normally you would install Docker on your development machine. If I got you to do this, it would be a can-o-worms, everyone running every conceivable system would ask me questions I can't answer. So, a better idea! We're going to use the micro-instance from project 2 as our (Docker not Java) development machine!

Raw, command line, unplugged. Hold my socially appropriate beverage!

INSTALLING DOCKER ON YOUR INSTANCE

1. Locate your existing instance - from your Project 2 AMI or make another fresh one from scratch. You are an expert now it'll only take a few moments - the dev machine will only need ports for ssh and 32000 open.

   If you reuse the Project 2 AMI you will need to remove the crontab and remove any existing server - or Docker will fail in step 7 because the port will be busy. **reboot** the instance after:

   ```
   % sudo crontab -r
   ```

2. Now we'll install Docker on our EC2 instance and make this our Docker "**dev**" machine.

   (a) Run a general update on your instance: **sudo yum update**

   (b) Install docker: **sudo amazon-linux-extras install docker**

   (c) Start Docker: **sudo service docker start**

   (d) Add the default user (so we don't need to be root and sudo everything): **sudo usermod -a -G docker ec2-user**

   (e) Log out of ssh using **exit** (to refresh the permissions we just set)

   (f) SSH back in.

   (g) Try the following command: **docker info.** You'll see a status dump, and you'll see no containers are hosted. We'll add some soon.

   - If that didn't work - then you did something above incorrectly. Go back and check.

If you can't find the key-pair that you saved for this instance, then I am afraid you can't recover the .pem file so you'll need to make a new instance and install...

- If you get the error: ***Cannot connect to the Docker daemon. Is the docker daemon running on this host?,*** try rebooting your instance in the EC2 dashboard.

MAKING A DOCKER IMAGE

Amazon ECS is able to use Docker images to launch containers on the instances in your clusters. So what we need to do next is use our Docker installation on the AWS instance to make a Docker image from your server.

3. If they’re not already on your instance, use SCP to copy your source files for your server to your AWS instance. The reason we need to do this is that **we're going to compile the java inside the container**. Believe me it is much less likely to go wrong this way! If you’re using Python, then they should already be there.

4. Now, what we need to do now is create a **Dockerfile** - this will be used by docker to construct an image. It contains environmental and build instructions. You need to either:

    - directly create a file containing the following codes, using say cat, or

    - upload it from your machine using scp.

5. You will need to substitute your program's name for mine - see the red parts. The file name is "Dockerfile" with no extension. You will also need to put the port in that your server uses. As usual mine is port 32000, so make that change to your Dockerfile. **Don't copy** the comments over - they're an explanation for this project. DO NOT JUST COPY AND PASTE - the encoding and hidden characters will break things, you will get weird errors!

    (a) JAVA DOCKERFILE:

```
FROM openjdk:8
COPY *.java /usr/src/M8B/
WORKDIR /usr/src/M8B
RUN javac Magic8BallServer.java
EXPOSE 32000
CMD ["java", “Magic8BallServer”,
“32000”]
```

```
// We are basing our image on the openjdk standard image
// copy our local files into a dir on the new docker image
// set the working directory
// execute the java compiler on our src file
// Make port 32000 on the container accessible
// Execute the container by running our java program
// passing in any command line args you need (port)
```

    (b) PYTHON DOCKERFILE:

```
FROM python:latest
COPY *.py /usr/src/M8B/
WORKDIR /usr/src/M8B
EXPOSE 32000
CMD ["python",
“Magic8BallServer.py”, “32000"]
```

```
// We are basing our image on the python standard image
// copy our local files into a dir on the new docker image
// set the working directory
// Make port 32000 on the container accessible
// Execute the container by running our java program
// passing in any command line args you need (port)
```

6. Now we have a Dockerfile, we can get Docker to build the new image - the name of the image needs to be lower case, incidentally m8b = Magic8BallServer:

    docker build -t m8b **.**

7. Assuming all is well Docker will now download the base image (remember it is immutable), update (add layers) and environment, and build your image. You might get

errors, for example if the name of your source file is wrong. They're mostly self explanatory.

8. Now we can check if Docker built the image properly, with:

   docker images --filter reference=m8b

   Mine looked like this:

```
$ docker images --filter reference=m8b
REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
m8b           latest    f5381436b32b   About a minute ago   526MB
```

9. Now, we can finally run the newly built image. Recall from the Dockerfile how we exposed port 32000 (or whichever one your application used). What we need to do now, is to map this exposed port on the container to a port on the host machine. In this instance we want to map 32000 to 32000, but we could map 32000 to a different port if we desired. We need to do this mapping so our application can interact with the outside:

   docker run -t -i -p 32000:32000 m8b &

   [1] 3482 <- you will get this response (different number) which says you have job 1 running in the background and its PID is 3482

10. Use **docker ps** to see if everything is running. I can't emphasise **enough**, any errors in your docker file, including funny characters - **will** cause the execution to fail.

```
CONTAINER ID   IMAGE    COMMAND                  CREATED          STATUS          PORTS
859973bebc93   m8b      "java Magic8BallServ…"   52 seconds ago   Up 51 seconds   0.0.0.0:32000->32000/tcp, :::32000->32000/tcp
```

   You can see the port mapping, the command, image and container ID. I deleted the names col.

11. You should now be able to run your original client and connect to the AWS instance, which will then tunnel the TCP connection to the container and your application. Your client should operate exactly the same as it did in Project 1. Mine looked like this:

```
~$java Magic8BallClient 3.94.21.212 32000 "should I eat this cookie"
Magic 8 Ball says: Don't count on it (172.17.0.2)
```

12. Congrats! You just containerised your first "app"

13. If you want to see a log of prior docker executions, you can use:

   docker ps -a

14. **[Don't do this unless you need to]** Incidentally you can also terminate a running docker image with:

   docker stop <CONTAINER ID>

CHECKPOINT:

What you have just done is the original way of hosting containers in AWS - were you explicitly used an EC2 instance to host - any number of containers. Each of the containers ports were mapped appropriately to ports on the host instance and the containers were supported and executed using the resources of the EC2 virtual machine.

However, this requires the developer to instantiate and manage EC2 instances or clusters of instances to host their containers, this also means small, ad-hoc containers that implement tiny-services like our server likely underutilise the EC2 instance, which means you're paying for resources you're not using.

PUSHING THE DOCKER IMAGE TO A CONTAINER REGISTRY

So far, we've made a docker image, run it and communicated with it, this is still a bit limited - because we're not taking advantage of the power of the container service. Indeed, we haven't yet told AWS anything about our image! It just looks like any other EC2 instance. What we want to do is put our Docker image into a repository, so that we can point AWS at it and then get AWS to manage, scale and deploy our containers for us. **Cool**.

CREATING YOUR DOCKER ACCOUNT

15. First things, sign up for a Docker account - *use lower case in your name (see below)*.

16. Go do this at: hub.docker.com

17. Validate your email

PUSH YOUR IMAGE TO THE DOCKER REGISTRY

To deploy a Docker image on AWS, it needs to be visible in a Docker registry. We can choose to publish our Docker image in any Docker registry we want, as long as AWS can reach it from the internet.

18. Log in to docker from your AWS development instance with:

    docker login

    WARNING, DOCKER DOES WEIRD THINGS WITH CAPITALS IN YOUR USER NAME. IT LOWER CASES THEM! IF YOU CAN'T LOG IN, ITS PROBABLY THAT!

19. You will need to enter your docker login credentials (the first time).

20. Use docker ps - to double check your joke server is running. We push running container images. (if not start one)

21. When you pull or run an image the name you give it is actually a location (URI) that also refers to the repository host. So you need to create an image tag that references your repo in Docker Hub before you push it to the repository. When we initially ran the image, we actually 'tagged it already' as 'm8b'. This tag isn't enough - we need to tag it with a tag that includes the repository. You can either tag it manually using docker tag, or run (in the background) it with the right tag. I'm choosing the first option (check your repository name on docker and replace yours in the following):

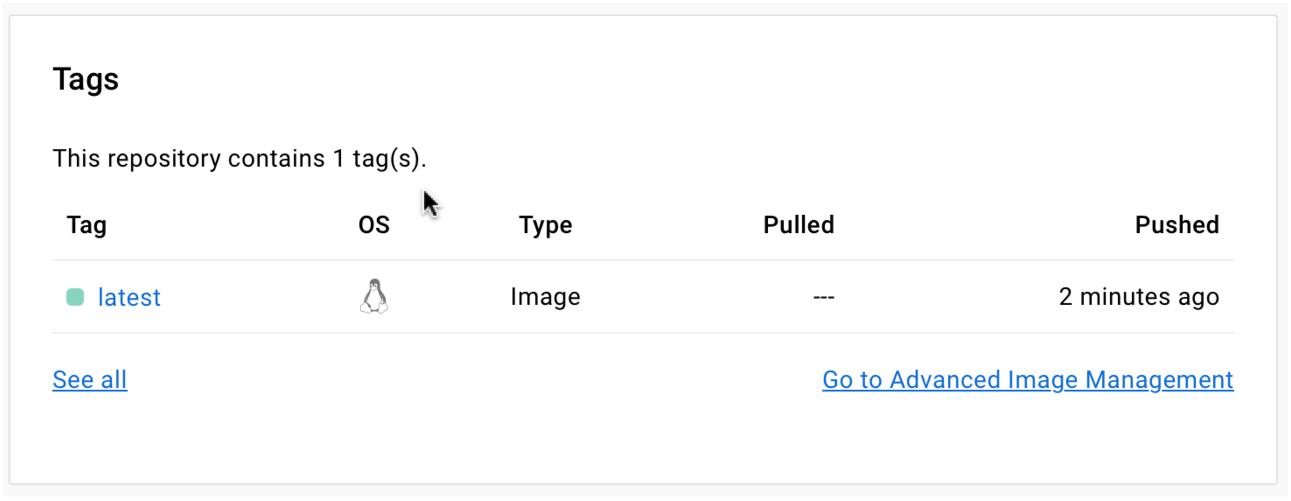
```
docker tag m8b  XXXX/m8b
```

22. push your image to the repository, obviously using your account and whatever name you built it with (docker actually tells you the cmd):

```
docker push XXXX/m8b:latest
```

If you don't have an exiting repository with this name "M8B" on Docker hub, the push command will automatically create that repository the first time you push.

23. It should now show up in your DockerHub. Well, after a little while anyway.



Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	---	2 minutes ago

See all

Go to Advanced Image Management

MAKE A TEST EC2 INSTANCE AND PULL THE CONTAINER.

24. Test your docker image - make a new vanilla EC2 instance and give it a suitable name, like "M8B Test" - so you can distinguish it from your dev instance. We're doing this so it is a clean install and we will know for certain our pushed image is correct. Just make a plain basic standard EC2 instance, remember to use the same Security Group ports.

25. Use the same keys.

26. You don't need to install java, but you will need to install docker (way back in step 2). You also need to start the docker service on the new test instance by following those instructions again. It goes without saying - there are no files in the directory…

27. Now pull the docker image you made - you can get this from your Docker repository screen:

```
docker pull docker.io/XXXX/m8b:latest
```

28. Now we run it a little differently, we still need the port map mine is:

```
docker run -t -i -p 32000:32000 XXXX/m8b:latest &
```

29. Now using your client (from before) you will be able to connect to the pulled docker image from docker hub on the new EC2 instance.

DISTRIBUTING YOUR CONTAINER - AWS ECS AND FARGATE.

Now we have a Docker container version of our Magic8BallServer - this is a standardised thing, so we can use other services to control and distribute our service. What we're going to do now is a **Serverless** version of the Magic8BallServer - using the Elastic Cluster Service and Fargate.

"AWS Fargate is a technology that you can use with Amazon ECS to run containers without having to manage servers or clusters of Amazon EC2 instances. With Fargate, you no longer have to provision, configure, or scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale your clusters, or optimise cluster packing. "

30. You'll probably have to type "ECS" (Elastic Container Service) into the search bar on the AWS console to find the right menus.

31. We need to start with a task definition. Choose task definitions and then "create new task definition"

32. Name your task (which will also name the name space - don't worry about that)

33. AWS Fargate (serverless) should already be ticked (selected)

34. Under "Task role" and "Task execution roles" select "lab role" from each of the drop downs - since the Learner Lab does not enable IAMS.

35. Scroll down to the container section. This is where you're going to supply the information for AWS Fargate to 'host' your docker container.

    (a) name your container

    (b) In the "image" box put the location of your Docker image from the Docker repository, something like docker.io/XXXX/m8b:latest

    (c) The default port for the container is port 80 (HTTP) you don't need this, so remove it as add a port mapping for your port - mine is TCP 32000, App protocol should be "none". You don't need a port name.

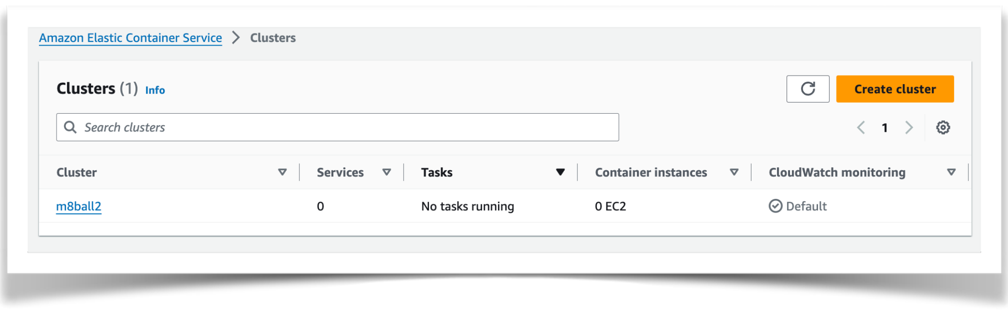
    (d) You can leave everything else as default.DEFINE YOUR SERVICE

36. You can leave the defaults on the Service Definition. You'll notice you could choose a load balancer, the number of tasks and security group, however, the load balancer is an application level balancer and would need HTTP for the health check and the security group that is created is exactly what we need.

YOU NEED SOMEWHERE TO RUN YOUR TASK - CREATE A CLUSTER

37. From the ECS menu, select "Clusters"

38. Now Create a Cluster - all you really need to do it name it, and make sure "AWS Fargate (serverless)" is selected (which is it by default). It can take a few minutes.



Amazon Elastic Container Service > Clusters

Clusters (1) Info

Create cluster

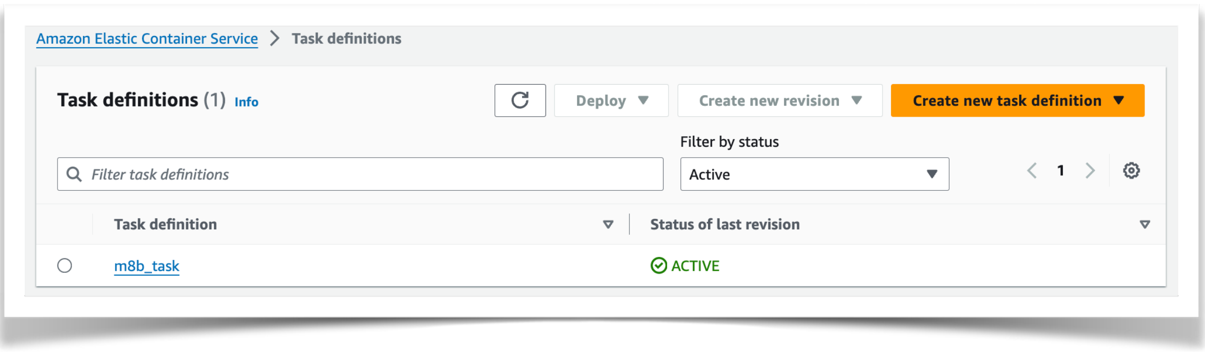
Search clusters

Cluster	Services	Tasks	Container instances	CloudWatch monitoring
m8ball2	0	No tasks running	0 EC2	Default

POPULATE THE CLUSTER

39. Now click on "Task Definitions" in the ECS menu.

40. You will see the task we made earlier that uses our Docker container.

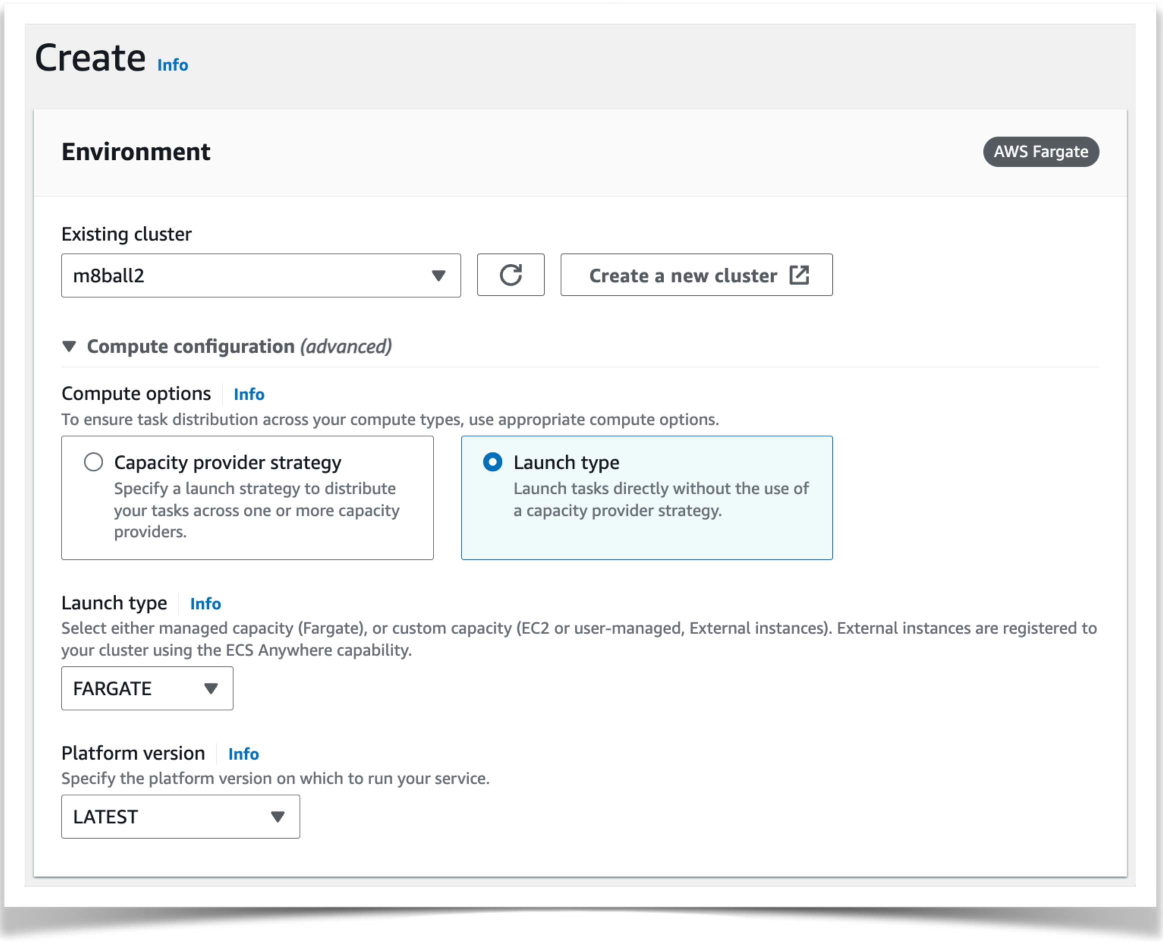

Amazon Elastic Container Service > Task definitions
Task definitions (1) Info
Deploy
Create new revision
Create new task definition
Filter by status
Filter task definitions
Active
1

Task definition	Status of last revision
m8b_task	ACTIVE

41. You will see the deploy button is greyed out - you need to click on the radio button next to the "m8b_task" in my example above.

42. Select create from the deploy dropdown.

43. We're going to make the simplest that we can. Ensure the cluster you just created is selected and then select "Launch Type" instead of the default.



Existing cluster: m8ball2

Compute options
To ensure task distribution across your compute types, use appropriate compute options.

Capacity provider strategy
Specify a launch strategy to distribute your tasks across one or more capacity providers.

[✓ Launch type]
Launch tasks directly without the use of a capacity provider strategy.

Launch type
Select either managed capacity (Fargate), or custom capacity (EC2 or user-managed, External instances). External instances are registered to your cluster using the ECS Anywhere capability.
FARGATE

Platform version
Specify the platform version on which to run your service.
LATEST

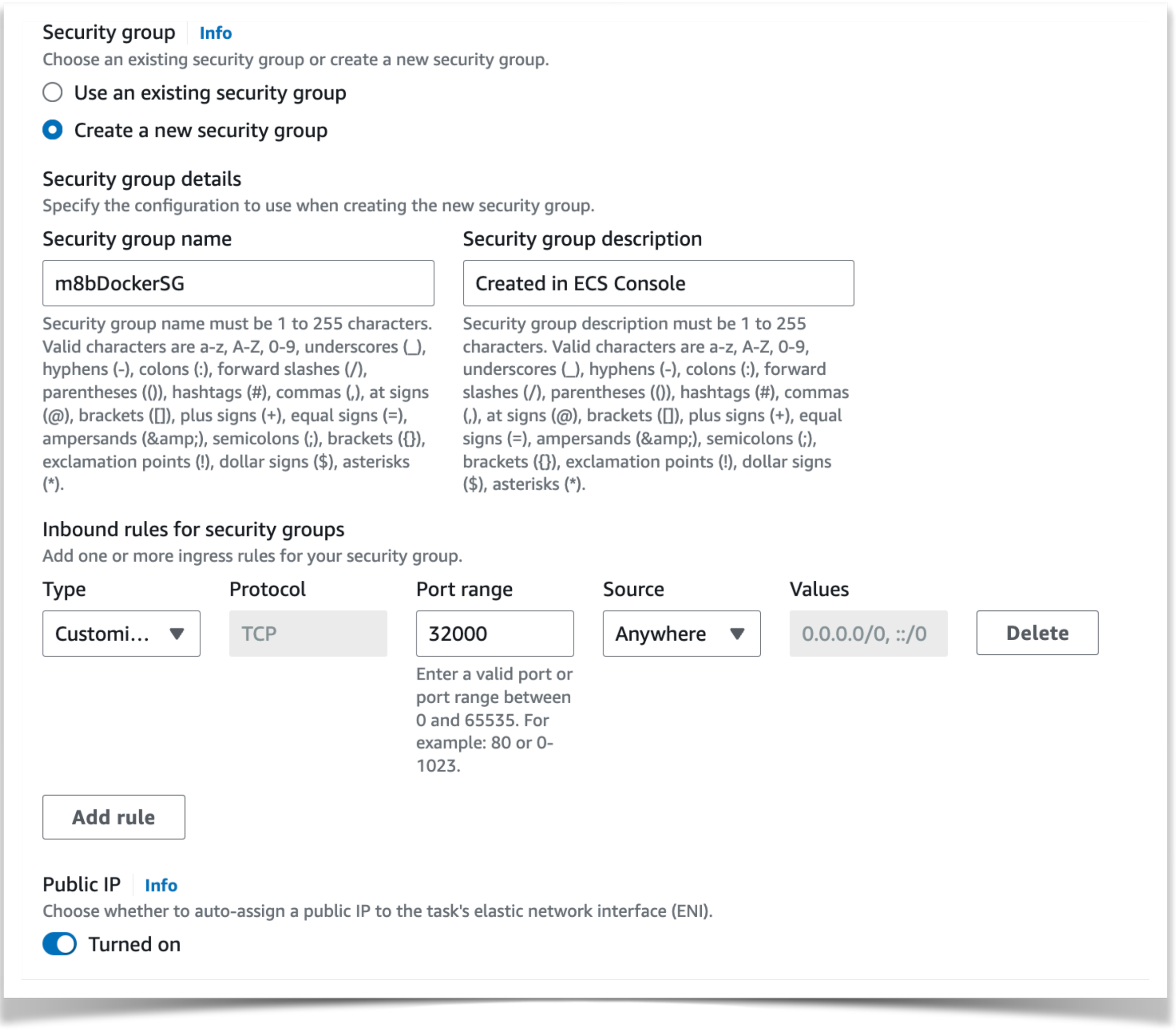
44. Scroll down on this screen to "Deployment Configuration"

45. Everything here is fine as default, you just need to fill in the name. Leave it as 1 task to launch. You're welcome to play around with this later, but for now - 1 is enough.

46. Next scroll down even further and expand the "networking" section.

47. Everything in here is fine, except the security group.

48. Create a new SG with our port open.



Security group Info

Choose an existing security group or create a new security group.

○ Use an existing security group

● Create a new security group

Security group details

Specify the configuration to use when creating the new security group.

Security group name	Security group description
m8bDockerSG	Created in ECS Console
Security group name must be 1 to 255 characters. Valid characters are a-z, A-Z, 0-9, underscores (_), hyphens (-), colons (:), forward slashes (/), parentheses (()), hashtags (#), commas (,), at signs (@), brackets ([]), plus signs (+), equal signs (=), ampersands (&amp;), semicolons (;), brackets ({}), exclamation points (!), dollar signs ($), asterisks (*).	Security group description must be 1 to 255 characters. Valid characters are a-z, A-Z, 0-9, underscores (_), hyphens (-), colons (:), forward slashes (/), parentheses (()), hashtags (#), commas (,), at signs (@), brackets ([]), plus signs (+), equal signs (=), ampersands (&amp;), semicolons (;), brackets ({}), exclamation points (!), dollar signs ($), asterisks (*).

Inbound rules for security groups

Add one or more ingress rules for your security group.

Type	Protocol	Port range	Source	Values	
Customi...	TCP	32000	Anywhere	0.0.0.0/0, ::/0	Delete
		Enter a valid port or port range between 0 and 65535. For example: 80 or 0-1023.			

Public IP Info

Choose whether to auto-assign a public IP to the task's elastic network interface (ENI).

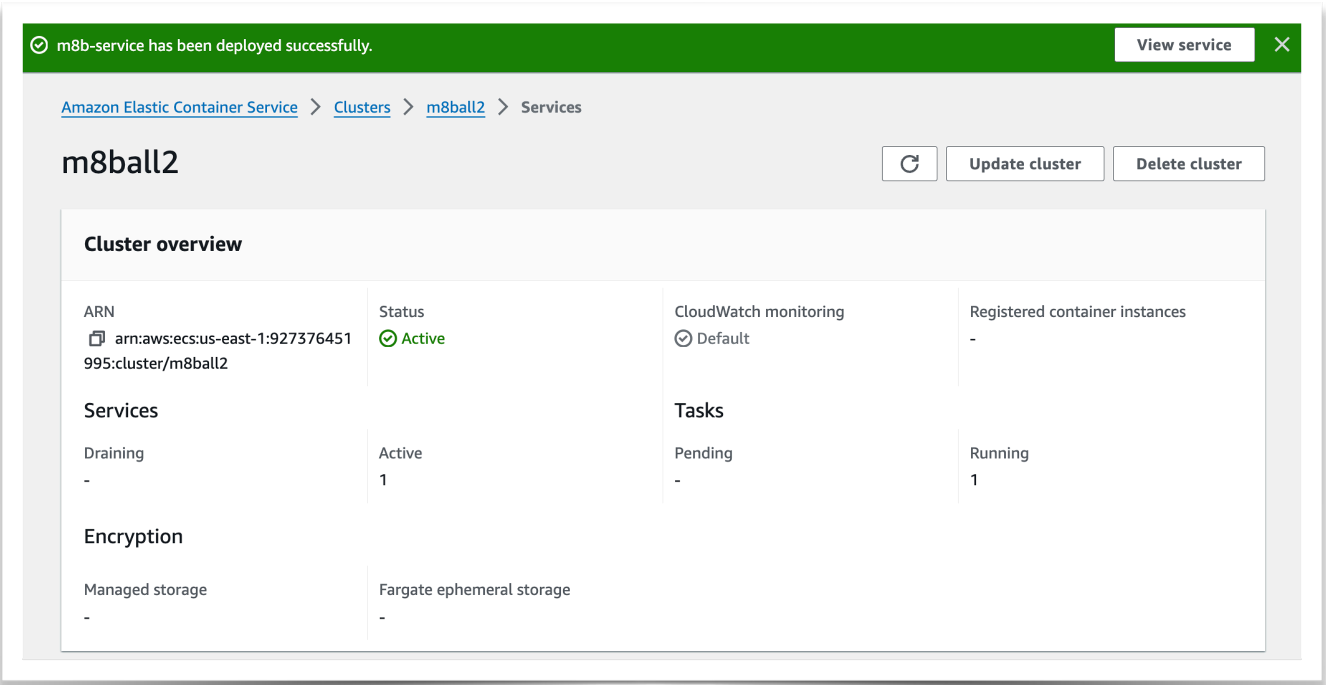
Turned on

49. You can see you can also configure load balancers, autoscaling, but let's leave that alone for this lab.

50. Now scroll the rest of the way down to the bottom of the page and click create.

51. You will need to be patient as it can take quite a while.

52. What you want is a screen that looks like this:


m8b-service has been deployed successfully.

Amazon Elastic Container Service > Clusters > m8ball2 > Services

m8ball2

Update cluster
Delete cluster

Cluster overview

ARN	Status	CloudWatch monitoring	Registered container instances
arn:aws:ecs:us-east-1:927376451995:cluster/m8ball2	Active	Default	-

Services

Draining	Active
-	1

Tasks

Pending	Running
-	1

Encryption

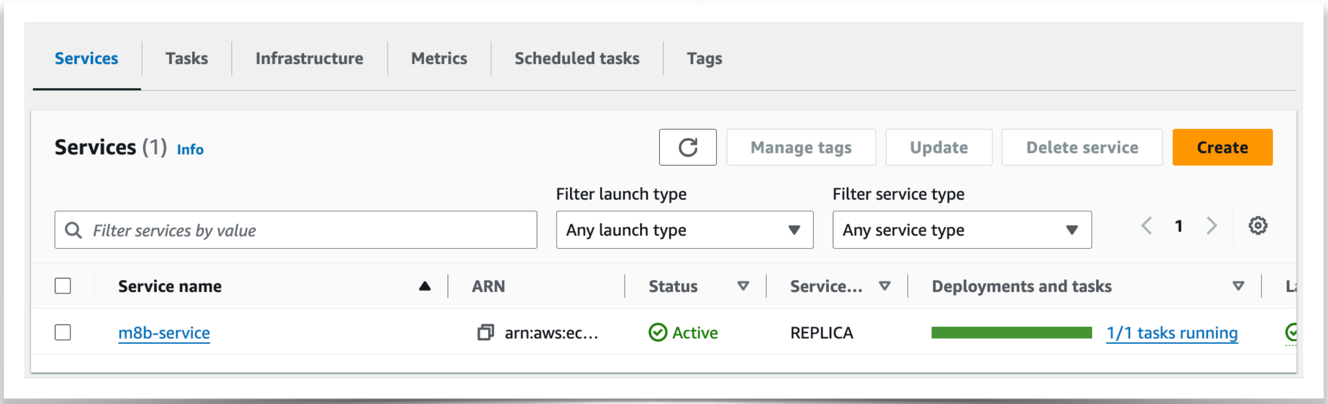
Managed storage	Fargate ephemeral storage
-	-

53. note the single active task and that it is running and no longer pending. This now means we can talk to it with our client.

    1. It is worth noting, if you have your repository wrong, you'll just notice the tasks tab, there may be nothing with a running status.

    2. You should then click on the stopped button, and if you see a lot of containers start and terminate with a "pull error" - thats your problem.

54. The services tab looks like this:



Services (1) Info

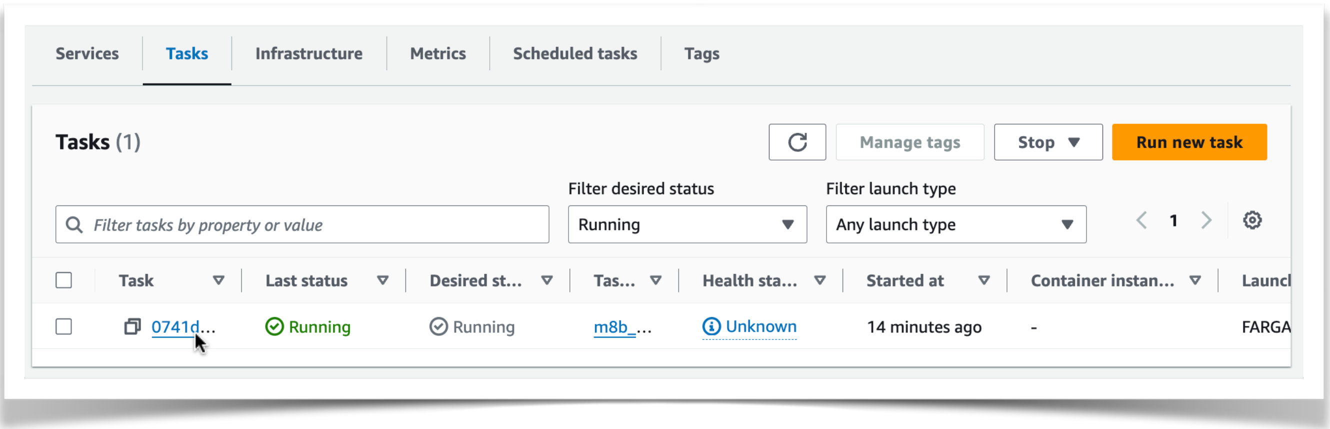
Filter launch type: Any launch type
Filter service type: Any service type

	Service name	ARN	Status	Service...	Deployments and tasks
	m8b-service	arn:aws:ec...	Active	REPLICA	1/1 tasks running

55. Assuming you have 1 running service, then you can test your service using your original client from anywhere - university, home, the bus.

56. We'll have to do some digging to find the host name for our client.

57. Click on the “Tasks” tab.



Tasks (1)

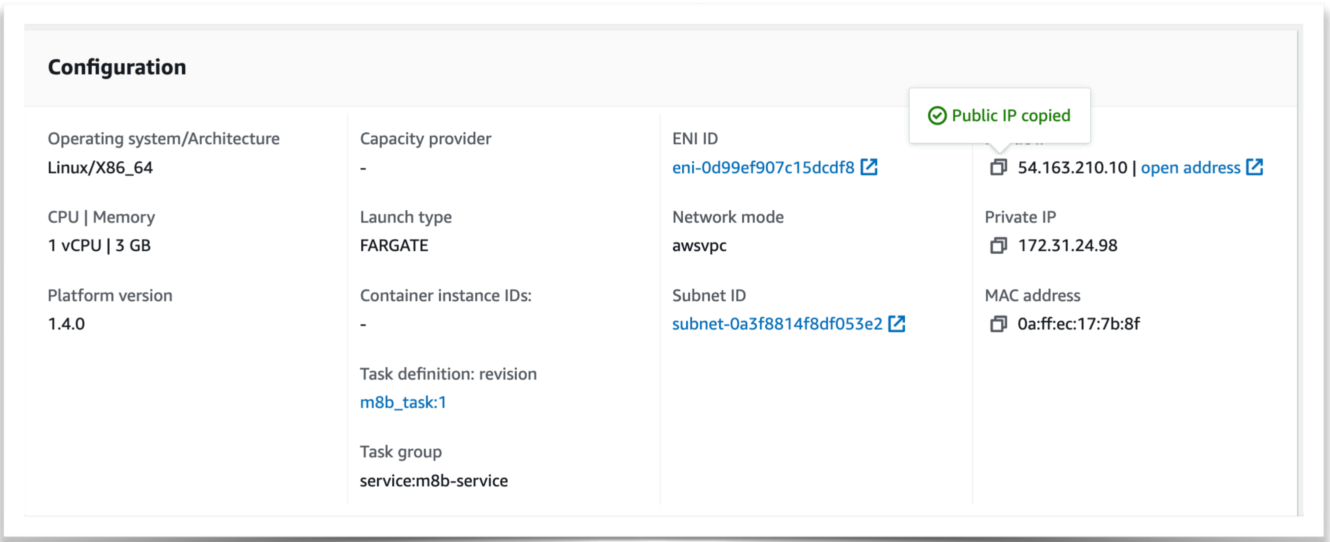
Filter tasks by property or value

Filter desired status: Running

Filter launch type: Any launch type

Task	Last status	Desired st...	Tas...	Health sta...	Started at	Container instan...	Launcl
0741d...	Running	Running	m8b_...	Unknown	14 minutes ago	-	FARGA

58. Then scroll down until you see the Configuration pane, which has the public IP address.



Configuration

Operating system/Architecture
Linux/X86_64

CPU | Memory
1 vCPU | 3 GB

Platform version
1.4.0

Capacity provider
-

Launch type
FARGATE

Container instance IDs:
-

Task definition: revision
m8b_task:1

Task group
service:m8b-service

ENI ID
eni-0d99ef907c15dcdf8

Network mode
awsvpc

Subnet ID
subnet-0a3f8814f8df053e2

54.163.210.10 | open address

Private IP
172.31.24.98

MAC address
0a:ff:ec:17:7b:8f

59. Copy the public IP and use it in your client.

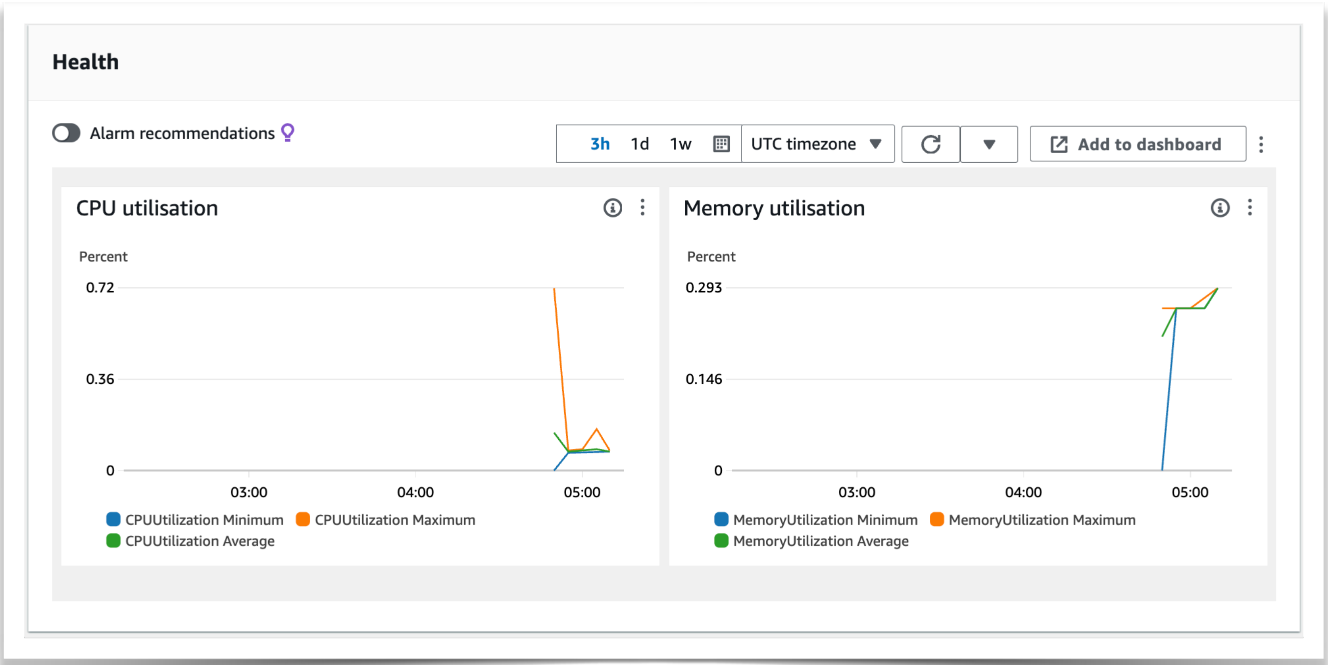
```
~$java Magic8BallClient 54.163.210.10 32000 "should I eat this cookie"
Magic 8 Ball says: Outlook not so good (172.31.24.98)
```

60. You now have your docker container answering your requests - serverless, in that you don’t set up or own the servers. Yes!

61. And it you want you can add more replicas, configure the loads, balancing etc.

62. Go back to the services tab for this cluster (ECS->Cluster->ServiceTab->service link) and scroll down to the health pane.

63. Here you can see the result of my single client call on the m8b service.



64. You can experiment with multiple/parallel call from the client etc. and you should be able to see the result here.

REPLICAS AND QUESTIONS

65. Your final task in this lab is to go back and create another cluster - but this time choose 2 or 3 tasks. Everything else should stay the same - except choose different names ok.

66. Question 1: What is unique about each replica - no **not** the ID. Although that **is** unique.

67. Question 2: How might we manage clients talking to replicas better?