Introduction to Containers

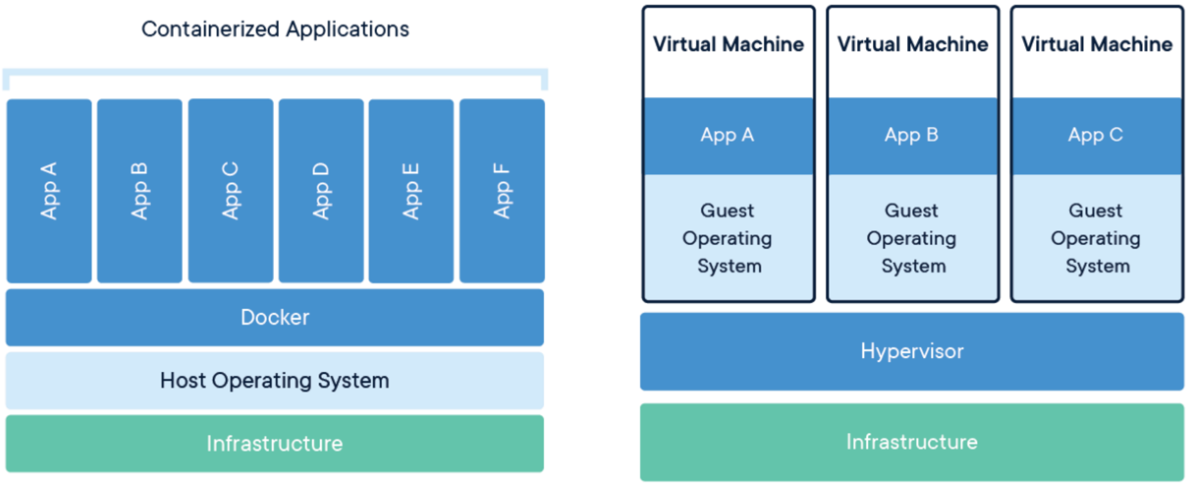


Fig. 1: A comparison of using containers versus virtual machines

Containerization is the standard in modern software development because it allows an application to run in a single package alongside all of its dependencies, libraries, runtime environment, and other configurations. This helps solve an age-old software testing problem, "But it works on my machine!"

Using virtual machines was common for this use case, however, virtual machines include an operating system. This adds to the total size of your application, increasing costs.

A containerization tool runs on the host operating system and manages your containers, making them more portable, and isolated, and reducing your total cost.

But how do we do this?

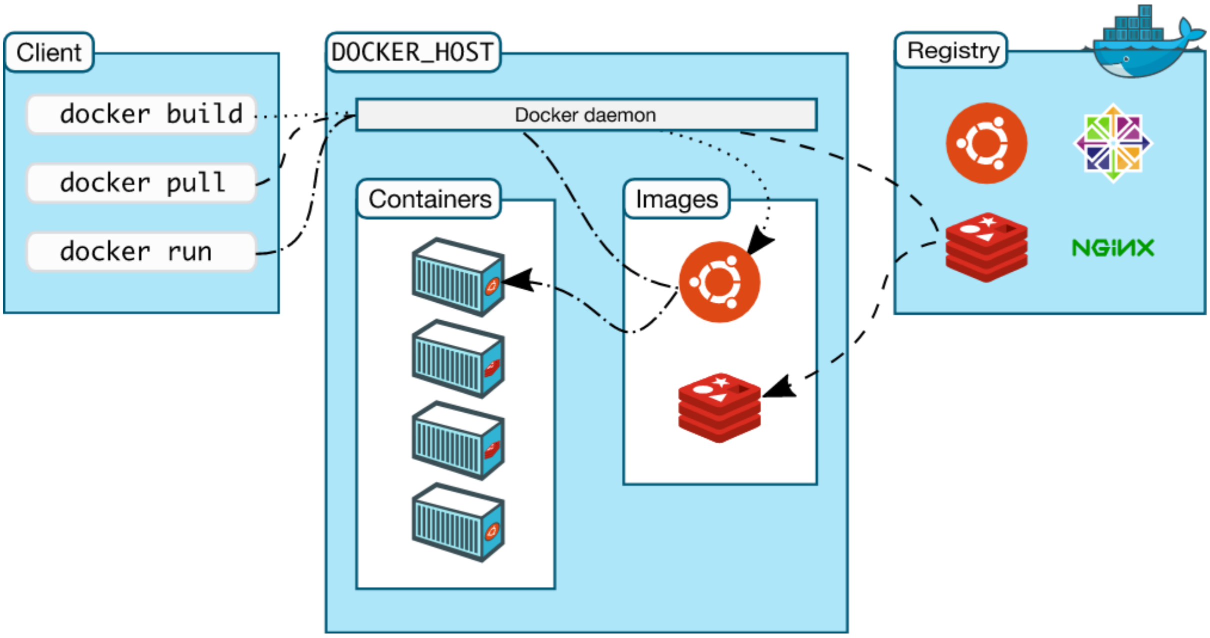


Fig. 2: Docker architecture

Docker is the most popular containerization tool. It allows you to package source code and other files into a Docker image, and then run a container from that image. A container is a **running instance** of that image.

In another perspective, think of a Docker image as the recipe for a cake, and a container as a cake you baked from it. The image spells out what ingredients should go into the cake. You cannot eat if you stop there. The container is the cake.

Docker images can be uploaded online to Docker Hub or other container registries such as **Amazon ECR** (Elastic Container Registry).

You can use commands on your terminal such as `docker pull`to install a docker image from Dockerhub, `docker build` to package your application into a container image using a **Dockerfile** (a file that specifies how your image will be constructed), and `docker run` to run a container from an image.

The Need for Container Orchestration

Running one container is easy, but what if you need to run ten, a hundred, or even a thousand of the same containers?

This is where container orchestration comes into the picture, which makes managing several containers much easier.

Common container orchestration tools include:

- Docker Compose
- Amazon Elastic Container Service (ECS)
- Amazon Elastic Kubernetes Service (EKS) / Kubernetes

In this workshop, I will provide an overview of Amazon ECS, its offerings, and a step-by-step walkthrough of deploying a containerized game on Amazon ECS.

Introducing Amazon ECS

Fig. 3: Amazon ECS Logo

Amazon Elastic Container Service (ECS) is a highly scalable fully-managed container orchestration service that helps you to more efficiently deploy, manage, and scale containerized applications. It deeply integrates with the AWS environment to provide an easy-to-use solution for running container workloads in the cloud that can range from hosting a simple web application to managing a distributed microservices-based architecture.

The benefits of ECS include:

- **Easy and simple deployment** — ECS eliminates the need to set up and maintain the infrastructure of Kubernetes clusters by taking responsibility for these tasks.
- **Scheduling capabilities** — that enable you to schedule services, applications, and batch processes.
- **Managed availability** — ECS is responsible for maintaining application availability and helps you scale up or down as needed to ensure capacity demands are met.
- **Native integration** — with a wide range of features like AWS ELB, Amazon Virtual Private Cloud (Amazon VPC), IAM, and AWS EBS.
- **Integration with existing tools** — ECS provides simple APIs that let you integrate with your CI/CD pipeline and your existing tools.

Core concepts and architecture:

Clusters:

Clusters are logical groups of container instances where tasks (containers) are run. A cluster can use either **EC2 instances** (Amazon EC2 launch type) or **Fargate** (a serverless compute engine) to host tasks.

ECS supports multiple clusters, and each cluster can scale independently, allowing you to manage the workloads of different teams or environments separately (e.g., dev/test/prod).

Tasks and Task Definitions:

Task Definitions are **blueprints** where you can specify how your containerized applications should run. They define important components such as:

- Docker images (Container runtime)
- Allocated CPU and Memory resources
- Networking configs
- Environment variables, storage, and logs
- Number of containers
- Data Volumes

Tasks can be instantiated from Task Definitions. Each task is a running instance of your containerized applications.

Services:

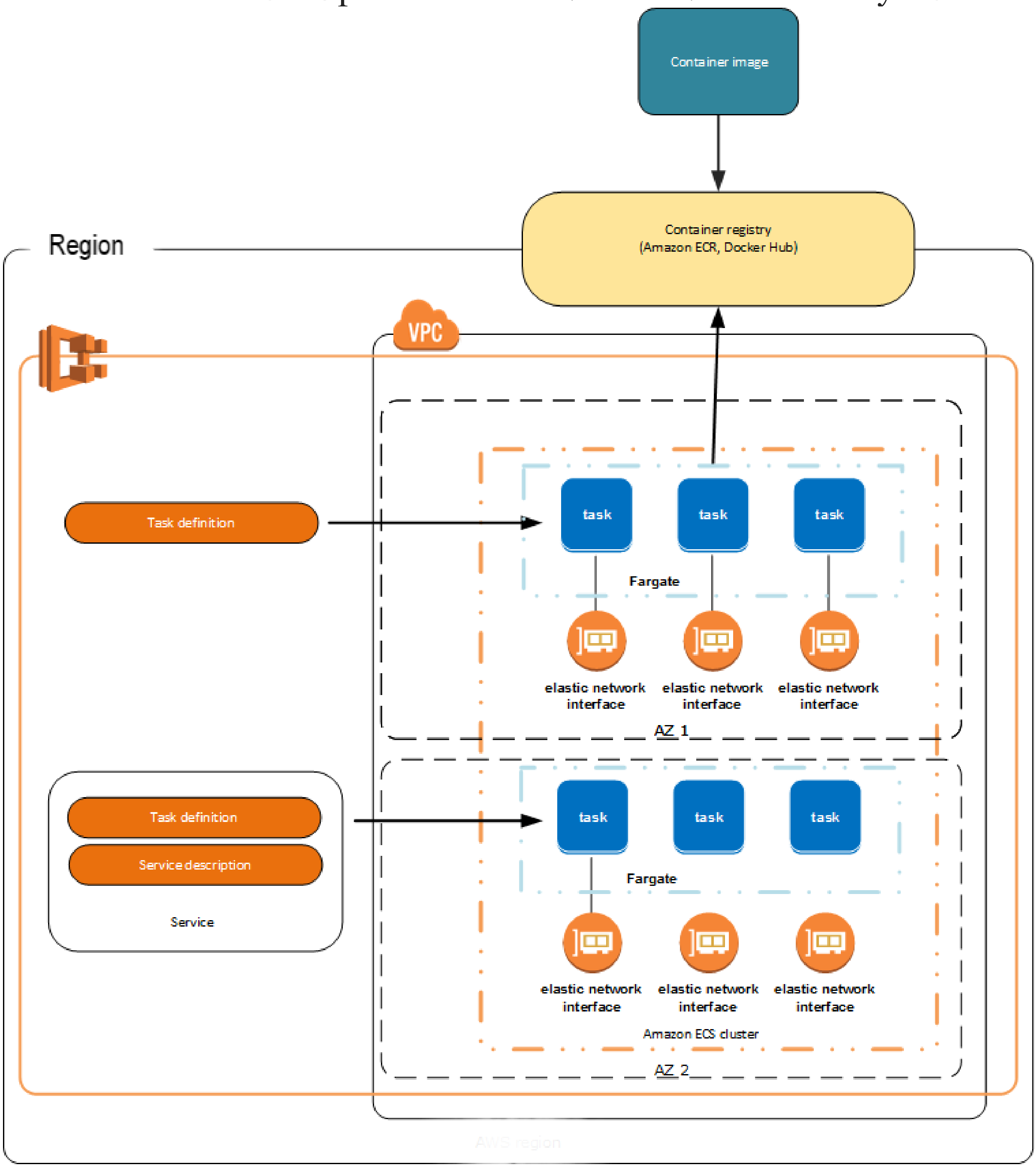
Services ensure that a specific number of tasks are always running and can provide useful functionalities such as load balancing, automatic failover, and auto-scaling.

- ECS integrates with **Elastic Load Balancer** to spread incoming traffic across tasks.

- Services can also be configured for auto-scaling based on policies or application demand.

Container Agent:

The ECS agent runs on EC2 instances to register the instances to an ECS cluster, communicate with the ECS API, and manage the task lifecycle. This agent is pre-installed on Amazon ECS-Optimized AMIs but can be run on any EC2 instance.



ECS Launch Types:

Amazon ECS supports two launch types for deploying and running containers.

With the **EC2 launch type**, you have full control over the underlying EC2 instances (container instances) that host your tasks. This launch type is ideal if you need custom AMIs, SSH access to EC2 instances, or Fine-grained control.

Key Features of ECS on EC2:

- Full control over compute, networking, and IAM.
- Ability to use reserved instances or spot instances for cost savings.
- Manage EC2 instances with auto-scaling groups.

AWS Fargate is a serverless compute engine for containers. It removes the need to manage the EC2 instances that your containers run on. Instead, AWS abstracts away the infrastructure layer, allowing you to focus purely on your application.

Key Features of Fargate:

- No need to provision or manage EC2 instances.
- Auto-scaling at the task level, Fargate automatically scales resources for individual tasks.
- Pay only for the compute resources and storage you use.
- Seamless integration with other AWS services.

Fargate is ideal when you want to **reduce management overhead** and need simple scaling.

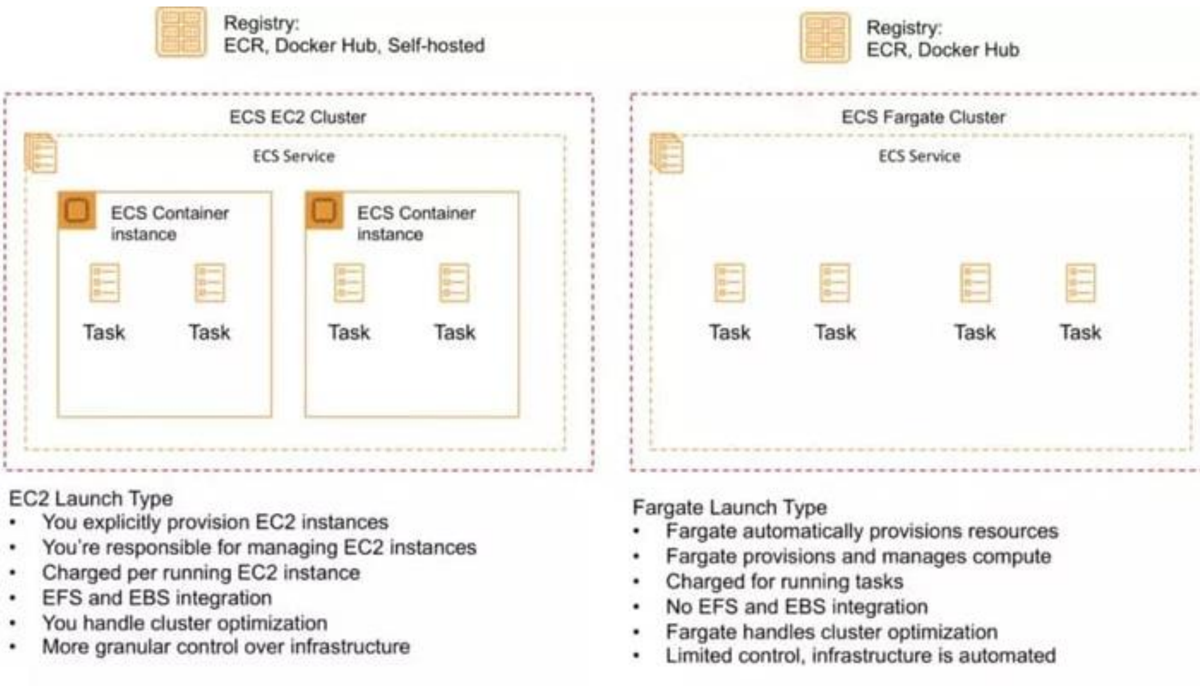


Fig. 5: Comparison between ECS Launch Types

Networking in ECS

Amazon ECS has flexible networking options to make sure your tasks have the right connectivity in your cloud environment.

VPC Integration:

ECS tasks can be launched in a VPC and are fully integrated with the Amazon VPC networking model. They can use all of VPC's features, including Security Groups, NACLs, VPC Peering, PrivateLink, and more.

Task-level networking:

ECS supports task-level networking using `awsvpc` mode, which gives each task its own **Elastic Network Interface** (ENI) and **IP address** in the VPC. This offers better control over network traffic and isolation between components.

Service Discovery and Load Balancing:

ECS services can be integrated with Application Load Balancers or Network Load Balancers to distribute traffic to tasks.

Service Discovery is also supported via Route 53. This allows ECS services to automatically register tasks to a DNS namespace, giving you the ability to discover and connect to your services.

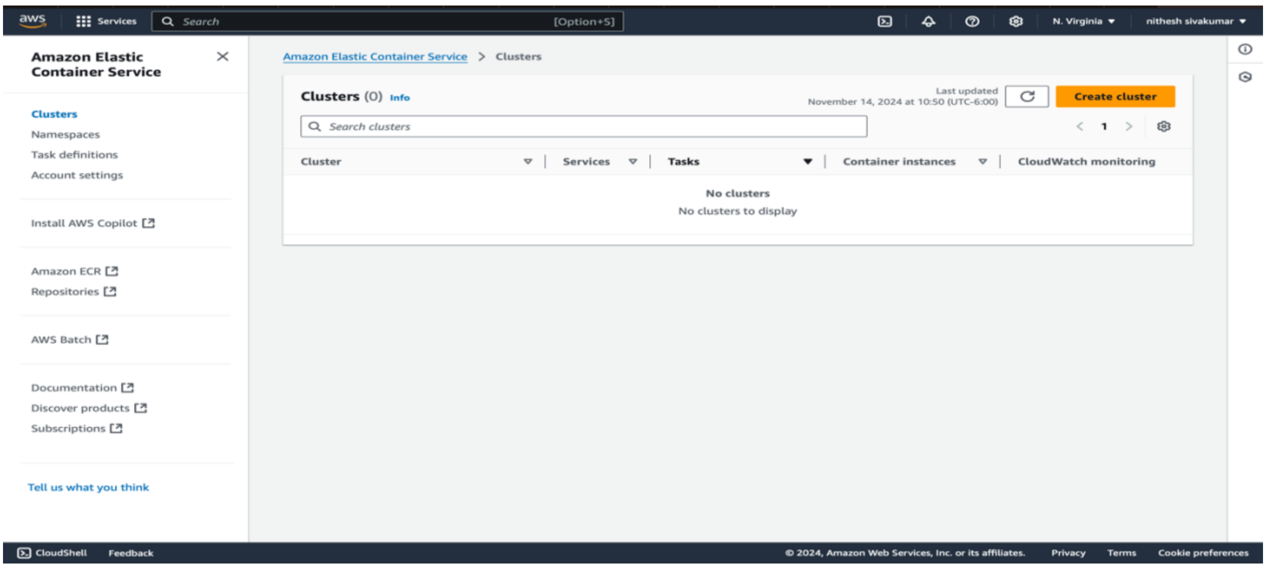
ECS Deployment Demo

In this demo, we are:

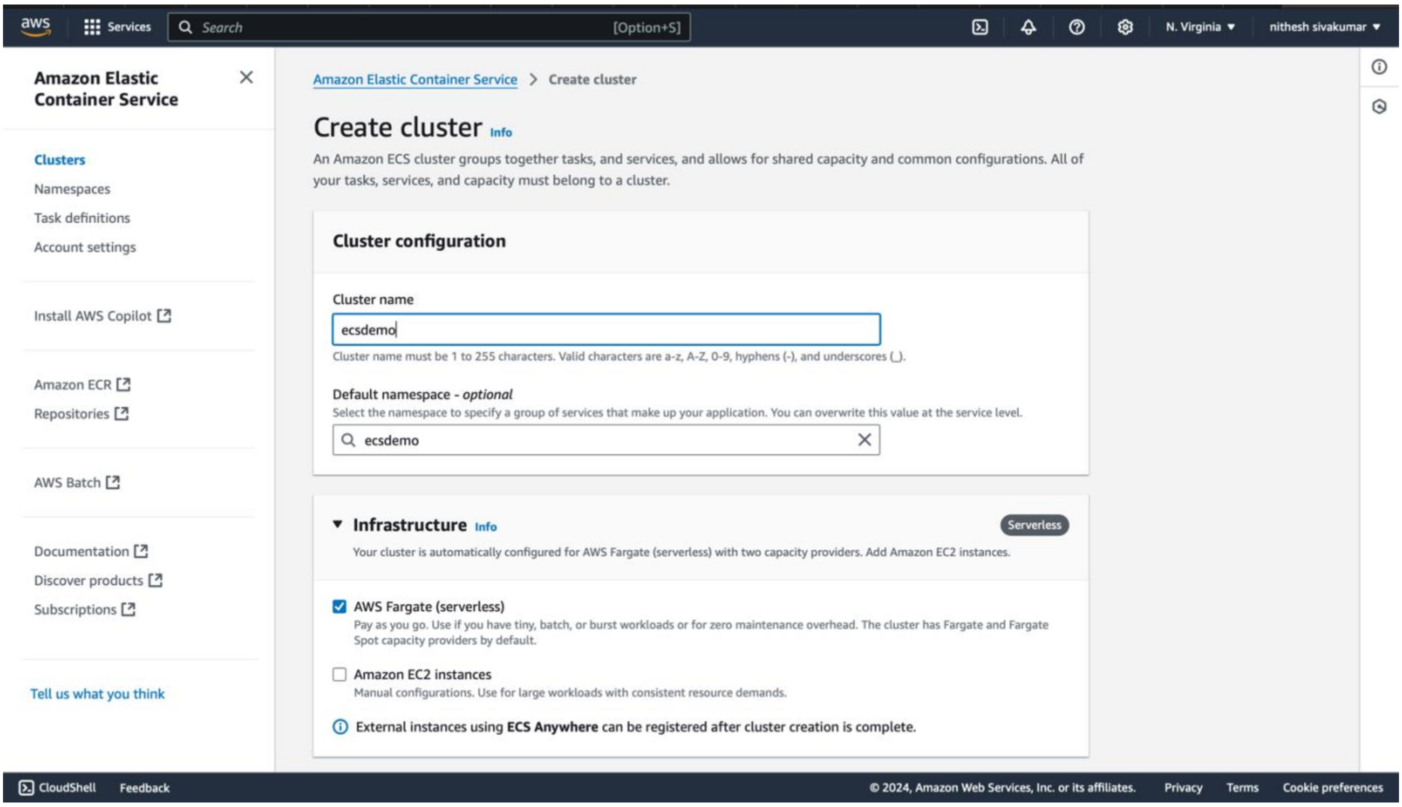
- Creating an ECS Cluster for Fargate
- Defining a task definition using this [container image]
- Launching a single task in a service, using that task definition

Task 1:

Ensure your region is set to US-East-1. Navigate to the ECS console.In the left menu, select clusters, then click "Create Clusters".



Input these options and keep the rest as default.

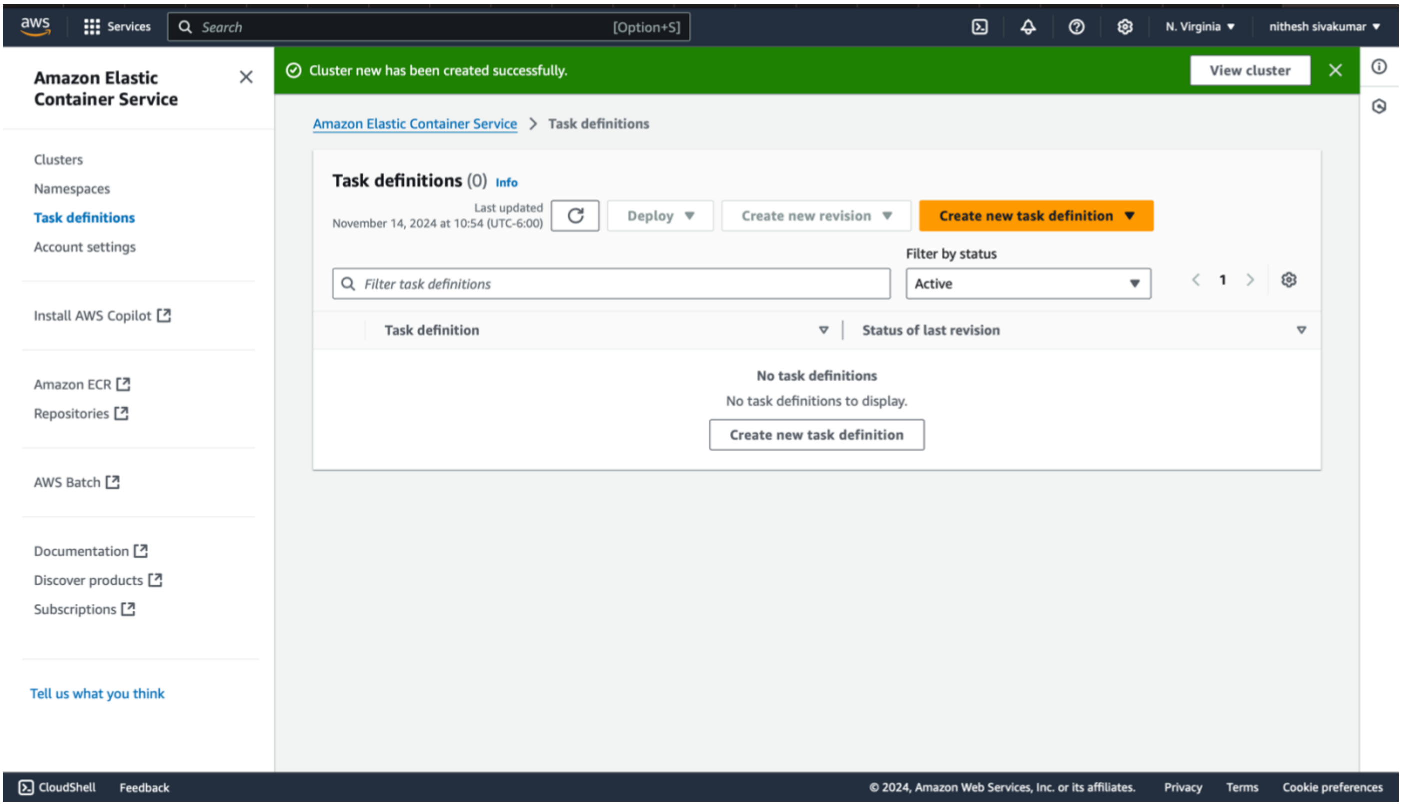


Scroll down to the bottom and click “Create”.

You may move on to Task 2 while the cluster is provisioning.

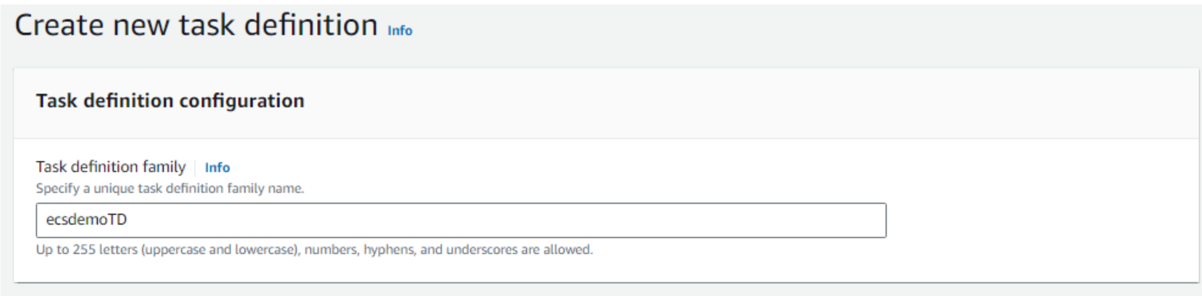
Task 2:

Move over to Task Definitions in the left-hand menu.



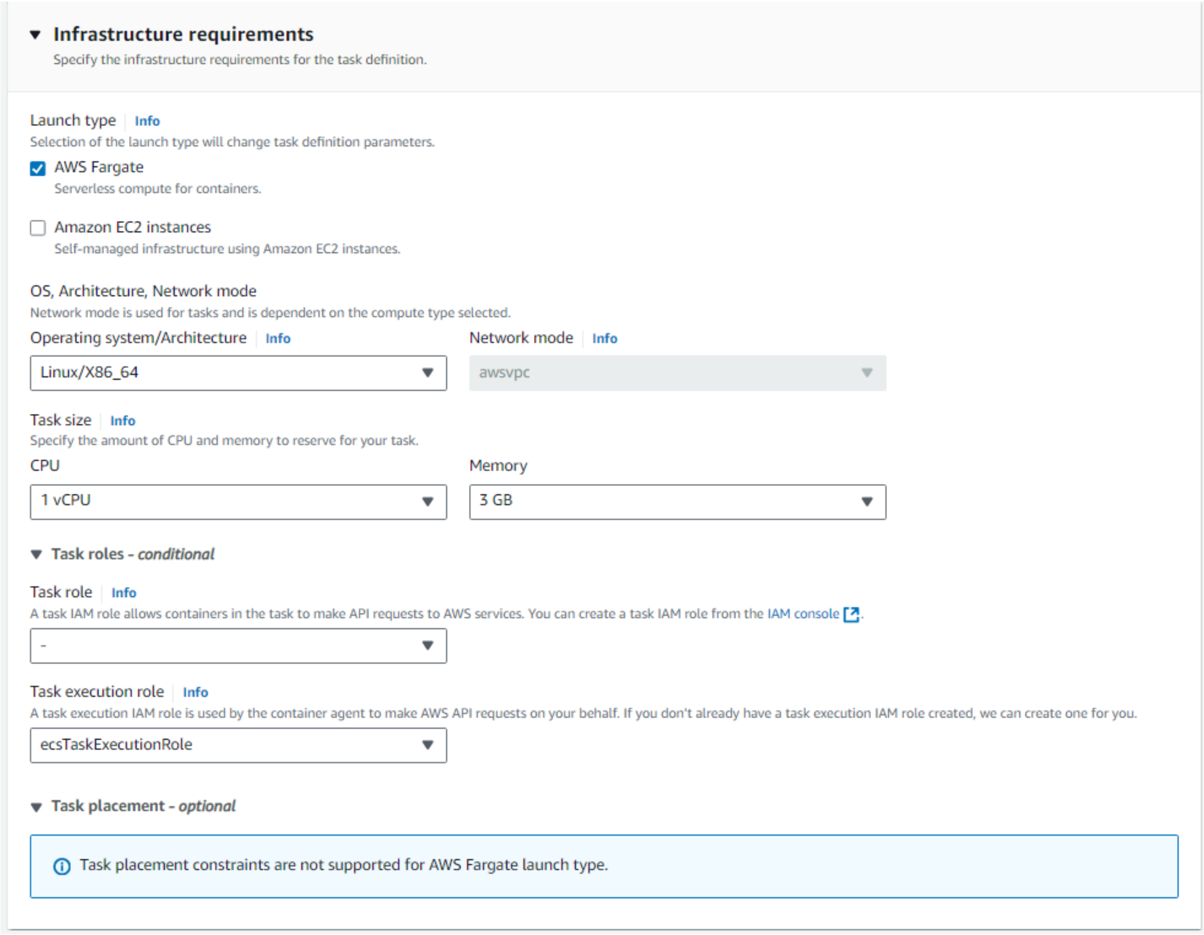
Click "Create new task definition".

Name your task definition anything you would like, for this example, I have used "ecsdemoTD".



Ensure you have AWS Fargate selected. The containers will run on top of a Linux x86_64 machine. For CPU and Memory, the default options should work fine because the container we run is not large.

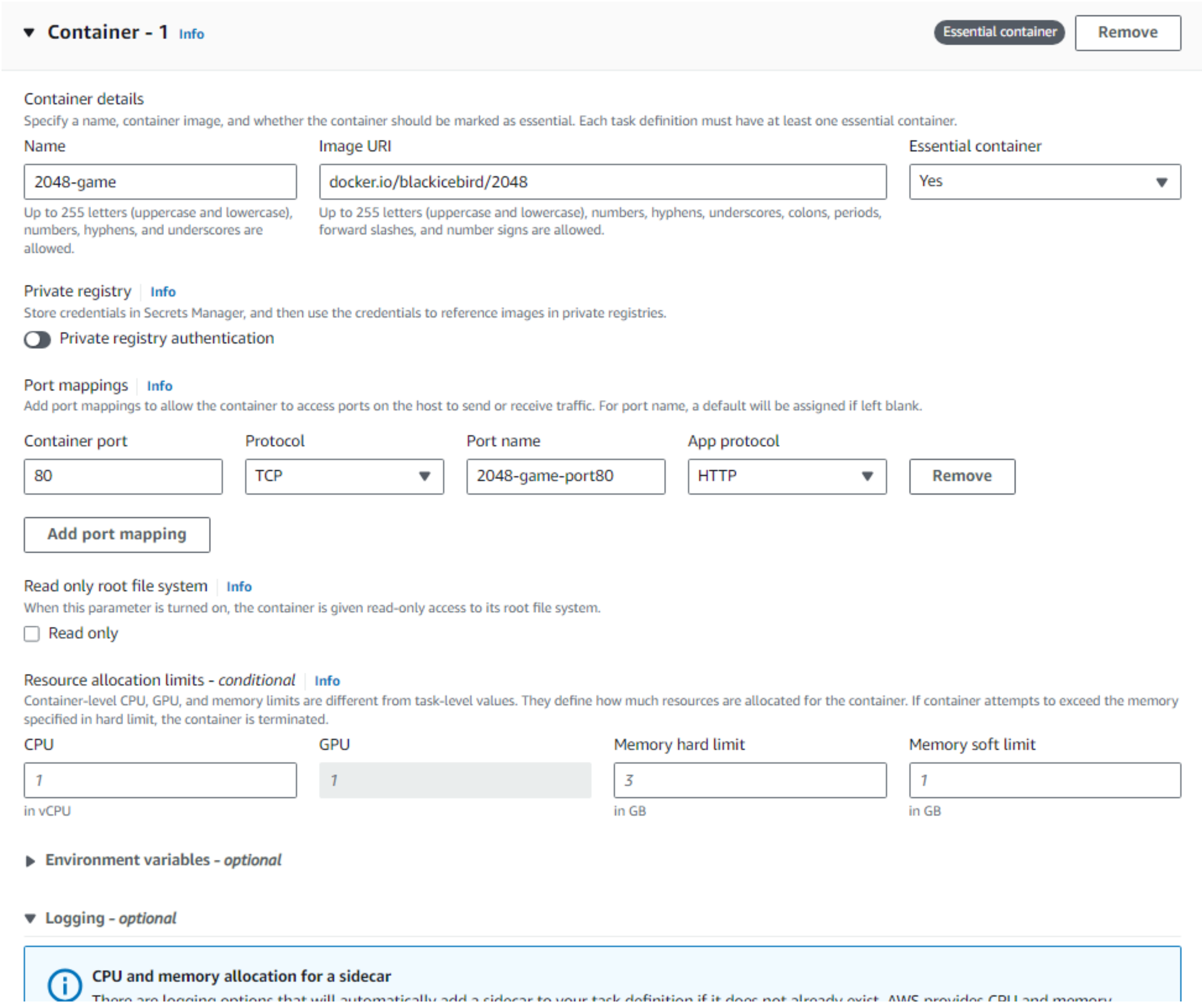
Under the Task roles section, leave Task role empty because we will not be making API requests to other AWS services for this demo. Ensure that you have a default IAM role called "ecsTaskExecutionRole". If not, AWS can create it for you.



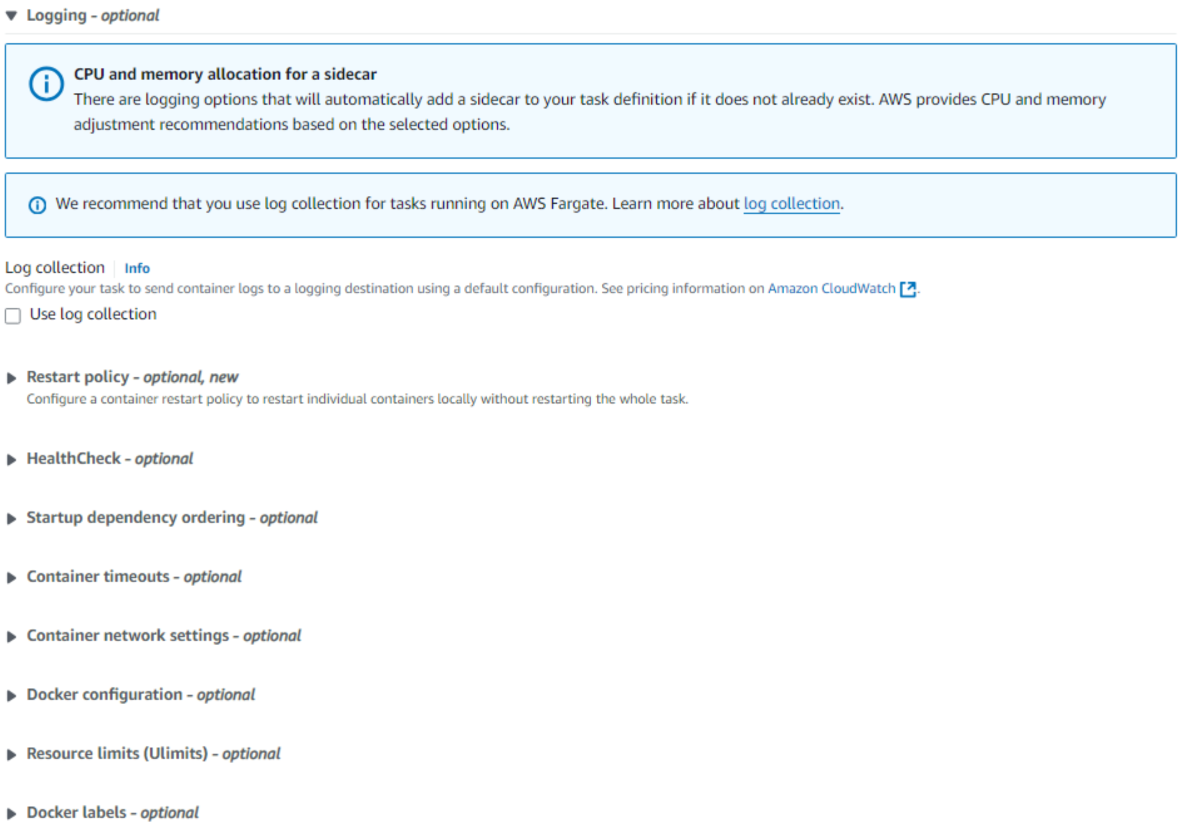
Follow the filled out fields for `Container-1`. This will ensure your container, named "2048-game" (or any other name you may choose) runs the docker image `blackicebird/2048`. This container is considered an essential container.

In ECS, an essential container is a key component of a task definition. An essential container must be completed for the task to be considered healthy. If an essential container fails or stops for any reason, the entire task is marked as failed. Essential containers are commonly used to run the main application or service within the task.

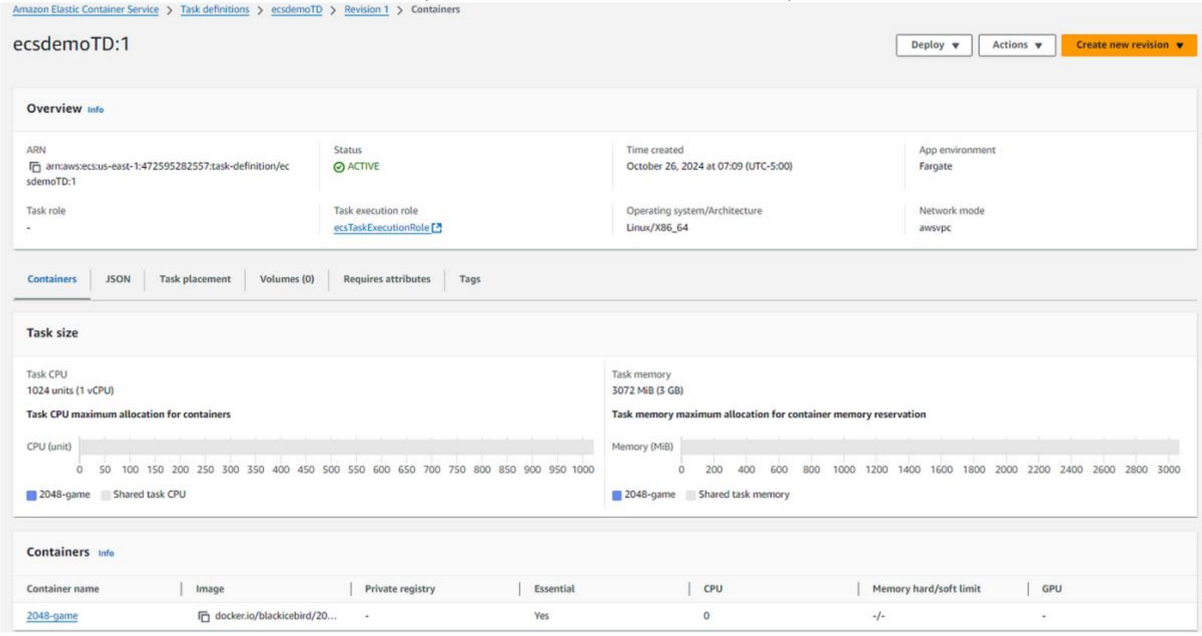
For port-mappings, the container will listen for HTTP traffic on port 80. For resource allocation, the default will work fine.



Uncheck logging because we will not be using CloudWatch for this demo.

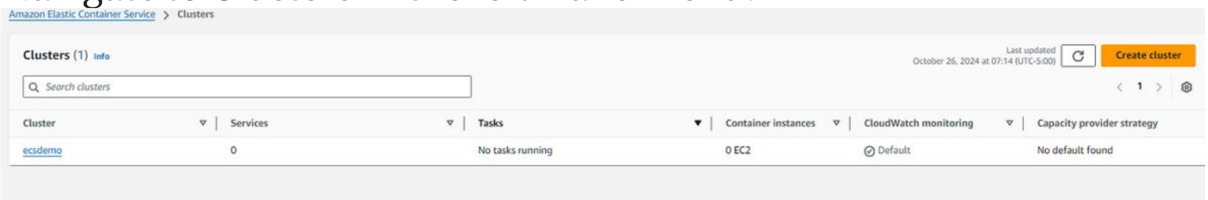


Leave the rest as default, scroll to the bottom, and click Create.



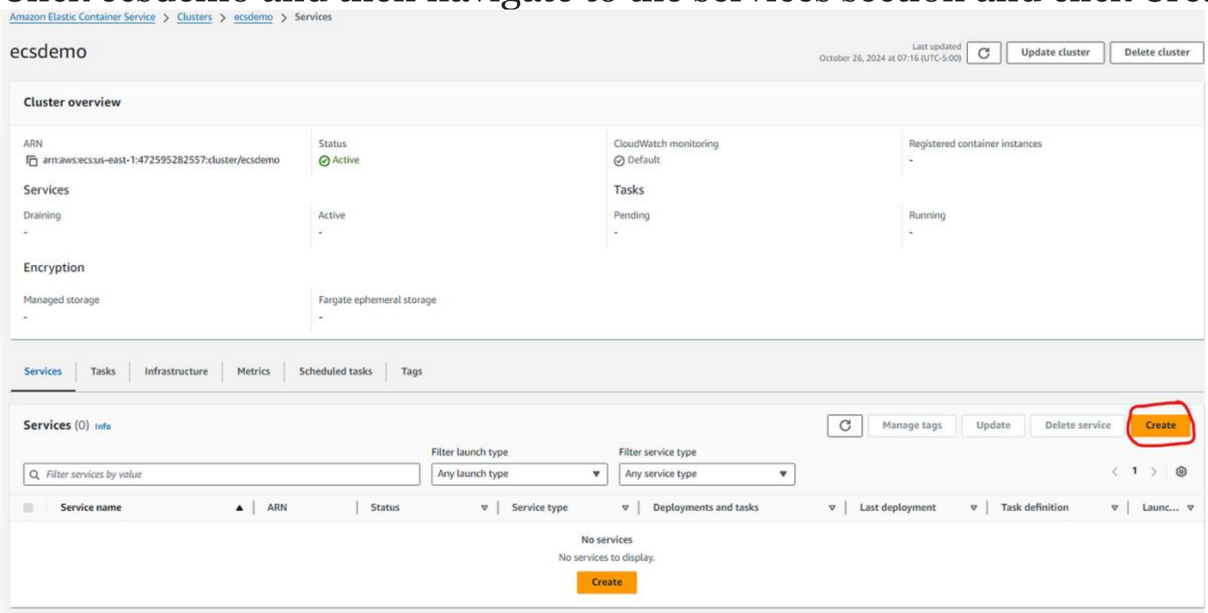
Task 3:

Navigate to Clusters in the left-hand menu.

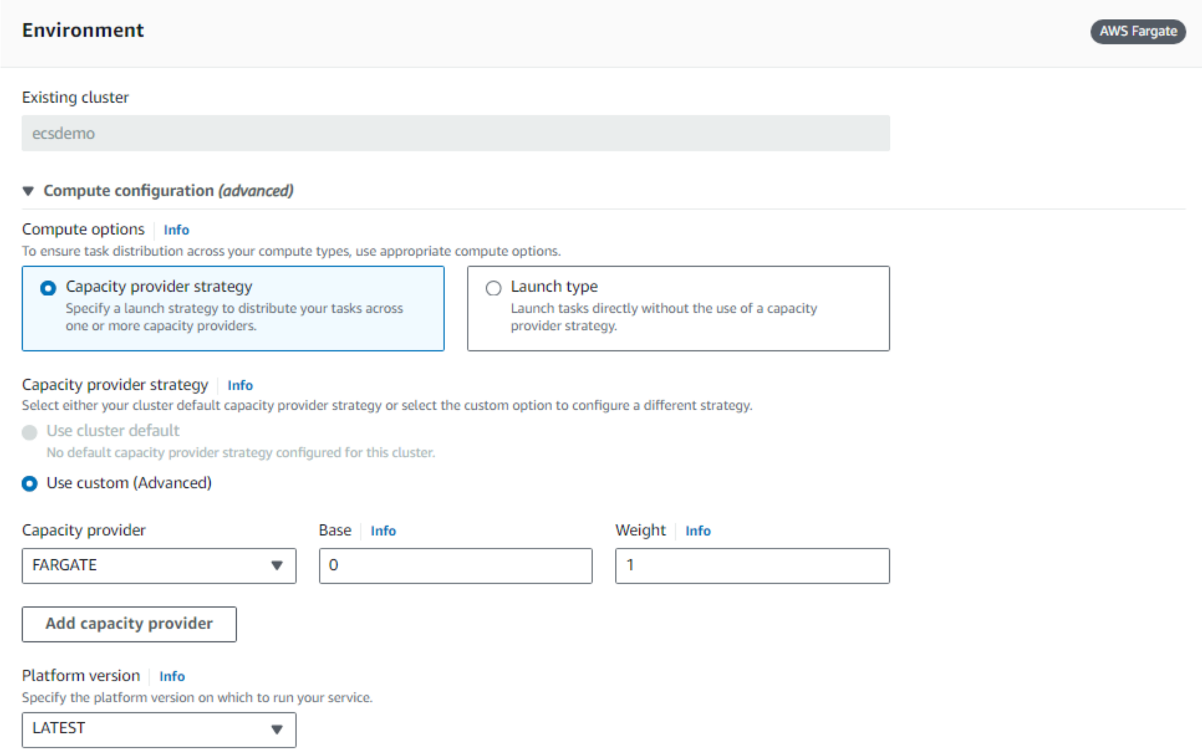


It's time to run our tasks!

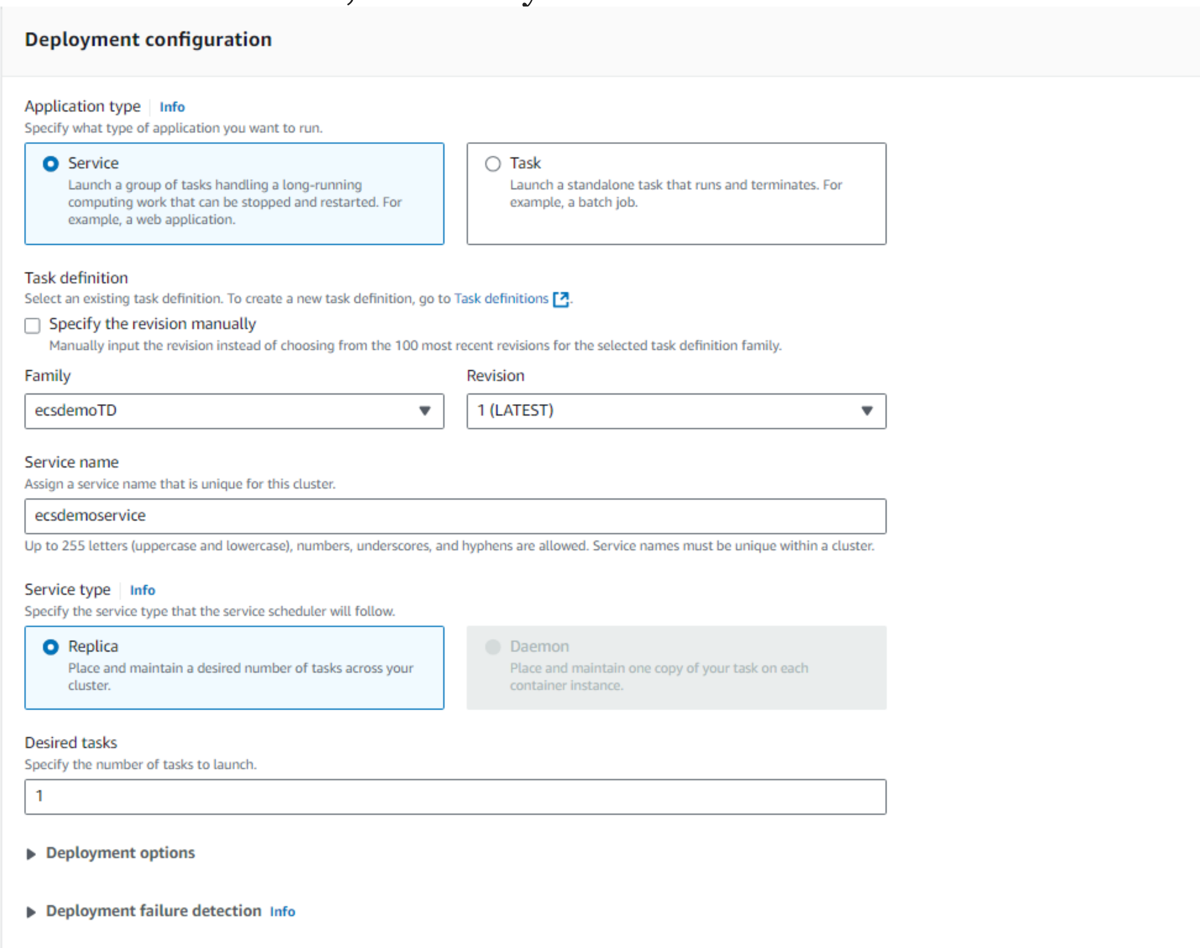
Click ecsdemo and then navigate to the services section and click Create.



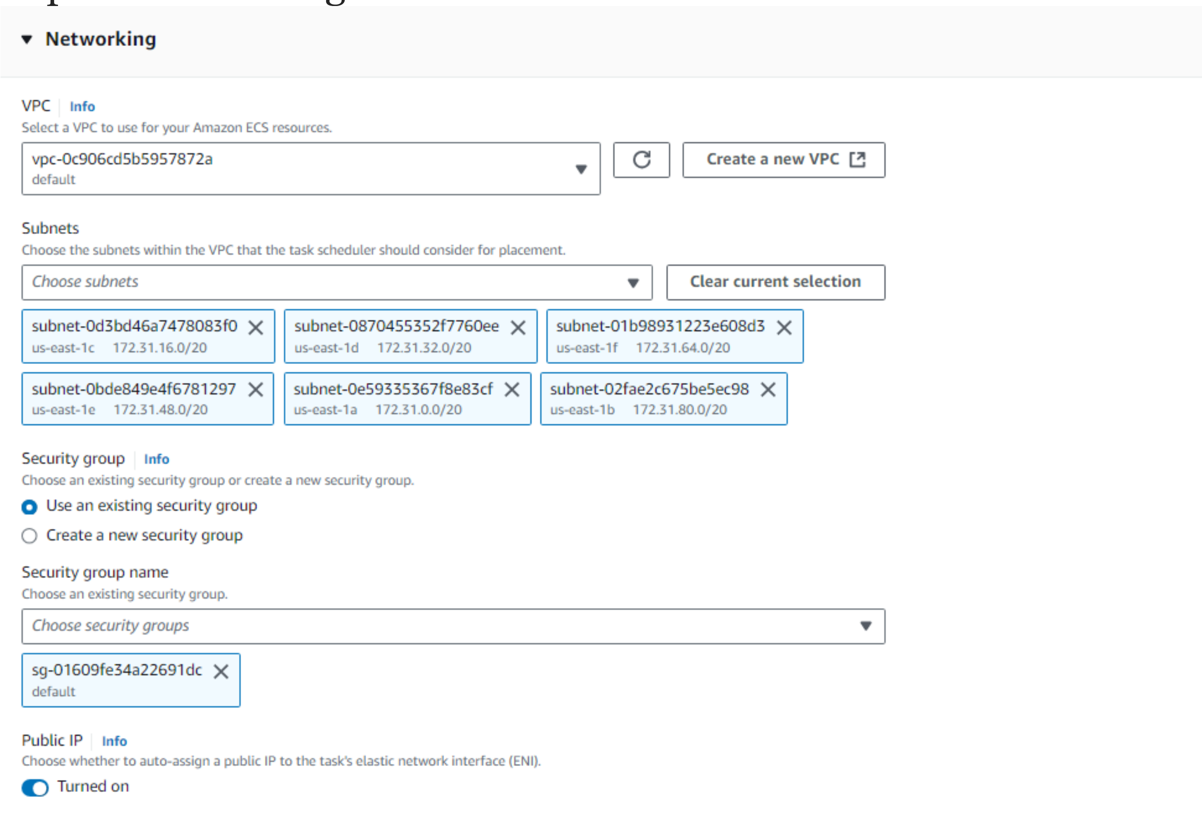
Leave environment as default:



For deployment configuration, ensure you have selected service for application type and set the values to what is listed below. If you named your task definition different from mine, then use yours.

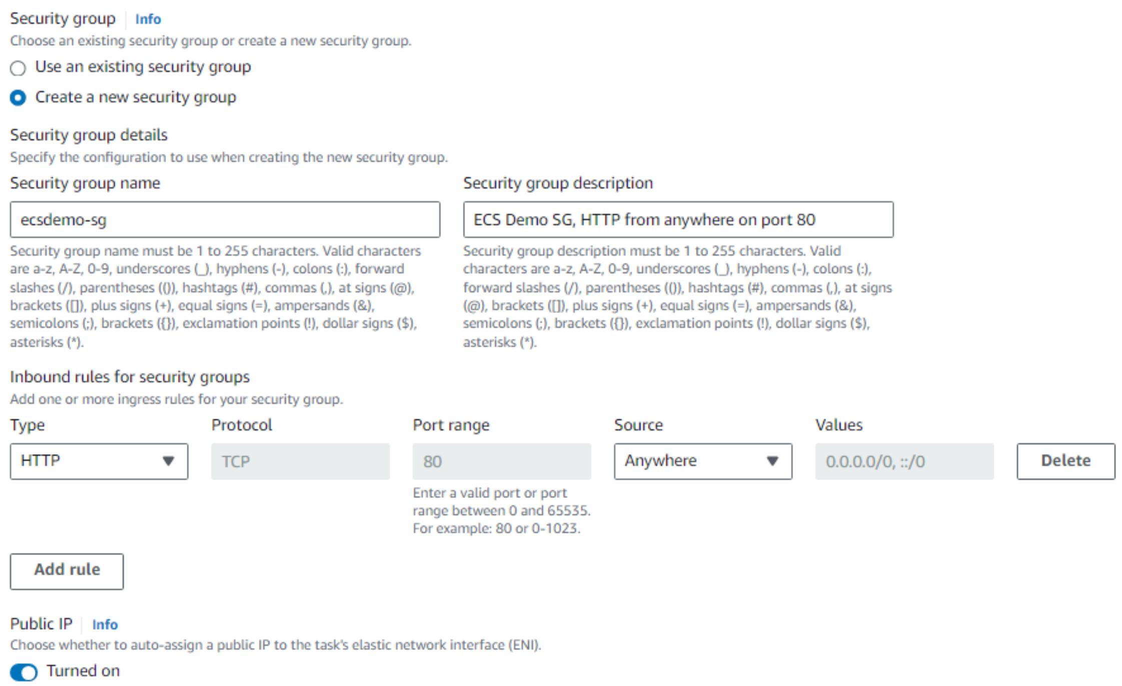


Expand networking:



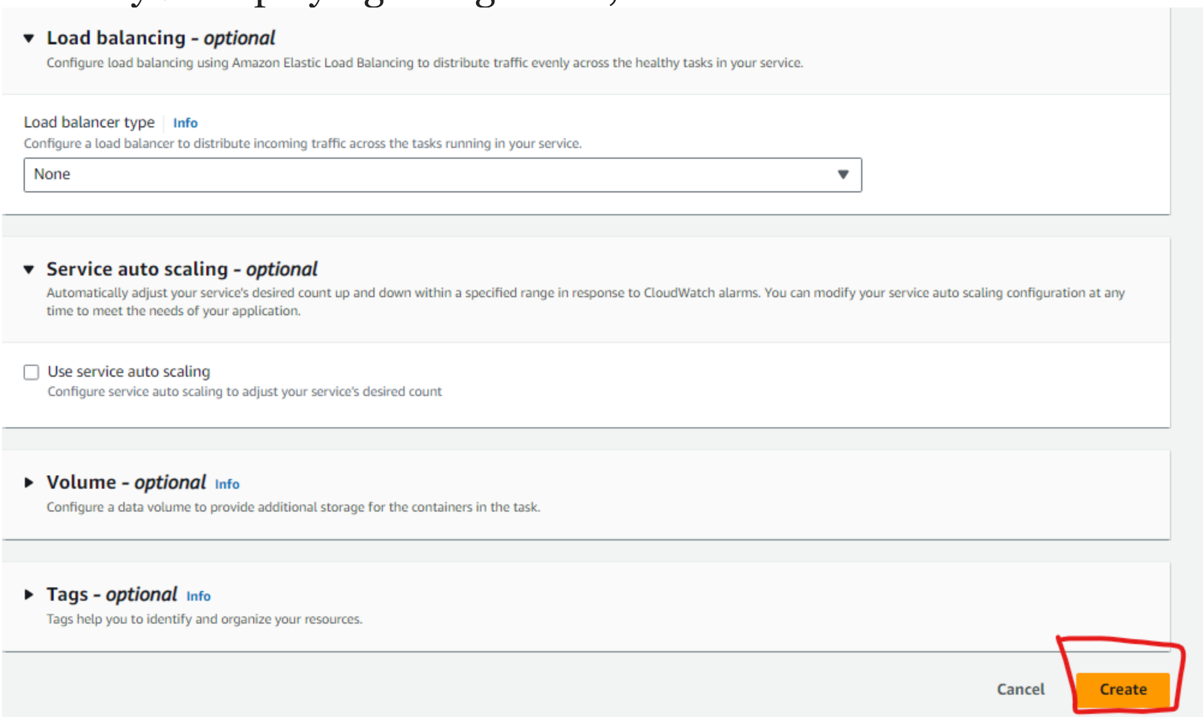
Ensure you have a default VPC, or create a VPC. Under "Security group", select "Create a new security group". Enter a name for your security group and a fitting description. Ensure that you have an inbound rule that accepts HTTP traffic on port 80 from any source.

Make sure your Public IP is turned on because we will be using this to connect to the task.



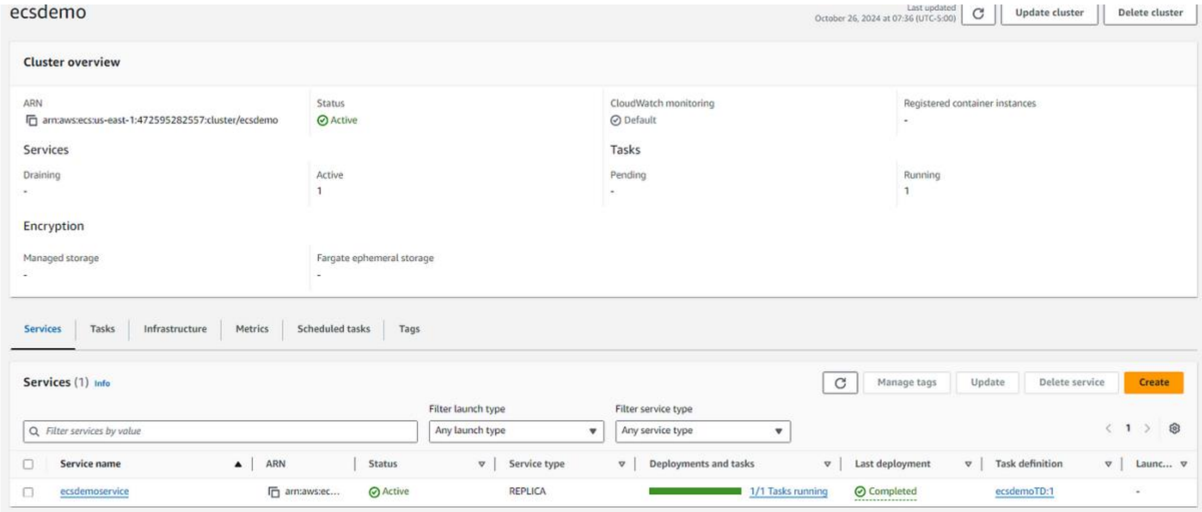
Review your configurations to make sure they are correct and scroll down.

For this demo, we will not be using a load balancer or auto-scaling because we will only be deploying a single task, so leave them unchecked and click Create.

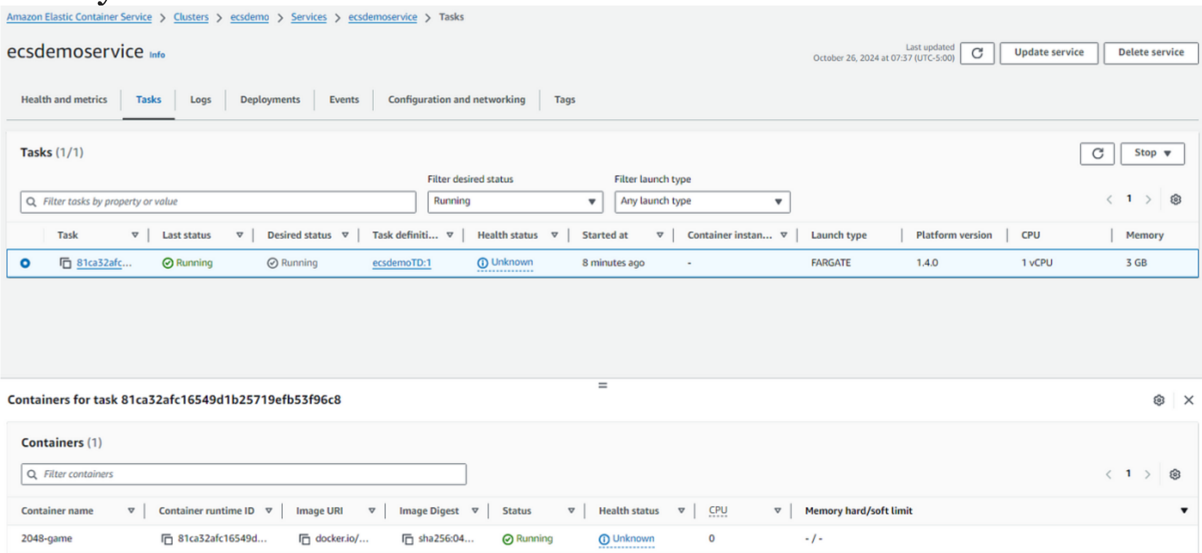


Wait until your service is deployed before moving onto Task 4.

Task 4:

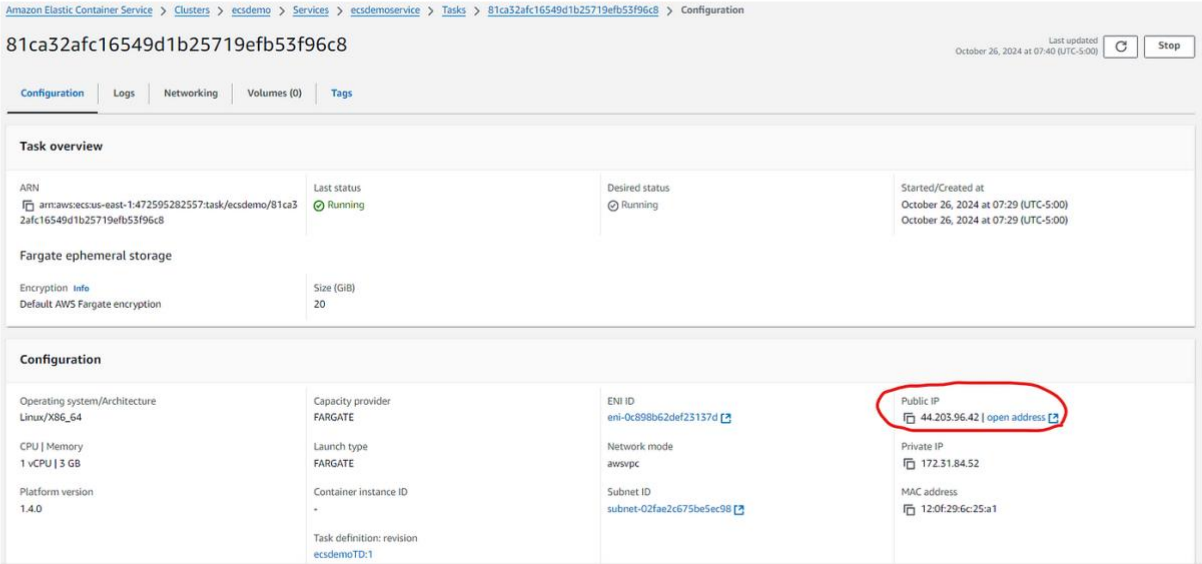


Your service should have Active status and 1/1 Tasks running. Click your service and move to the Tasks tab.

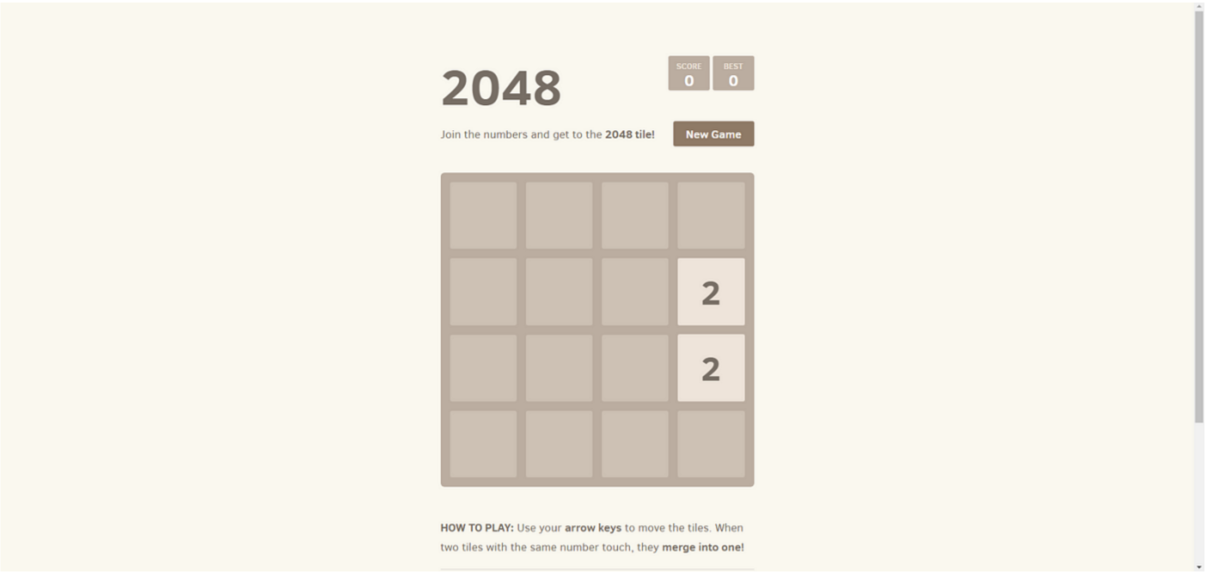


Here you can see your task's details, like Status, Health, CPU, Memory, Launch type, Image URI, etc.

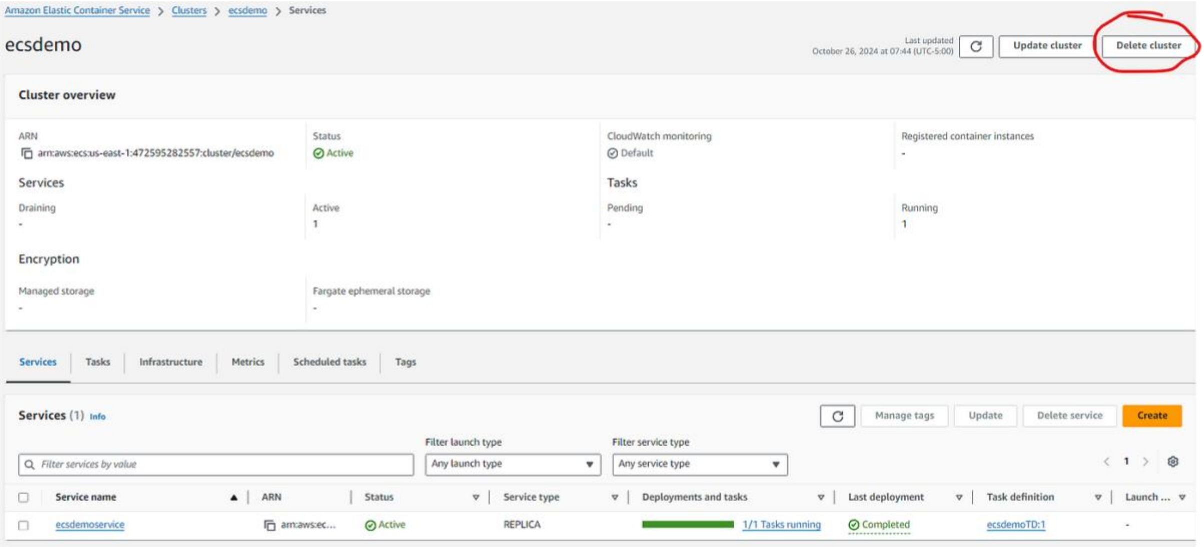
Click your task to view its configurations.



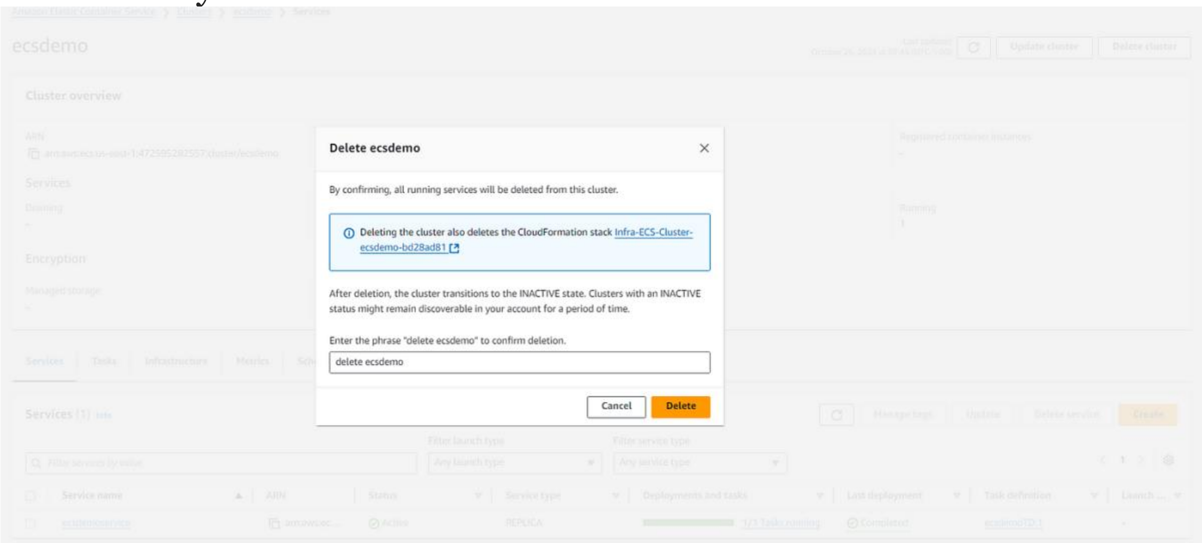
Copy the public IP into your clipboard, and paste it into a browser. You should see this:



You can play the game however long as you wish, but, you should delete your changes as soon as possible to keep your AWS bill low.



Go back to your cluster and select Delete Cluster.



Follow the instructions and click Delete. This concludes the demo, congrats!

Stopping the cluster will prevent you from being further charged. You don't need to delete your Task Definition because it won't charge you. However, you may do so if you desire.

Conclusion

Following through this article, you have understood the basics of Docker containerization, the need for container orchestration, and a practical understanding of getting started with Amazon ECS.