Creating a CI/CD enabled web application using AWS ECS Fargate and Code Pipeline

Jason Hoog

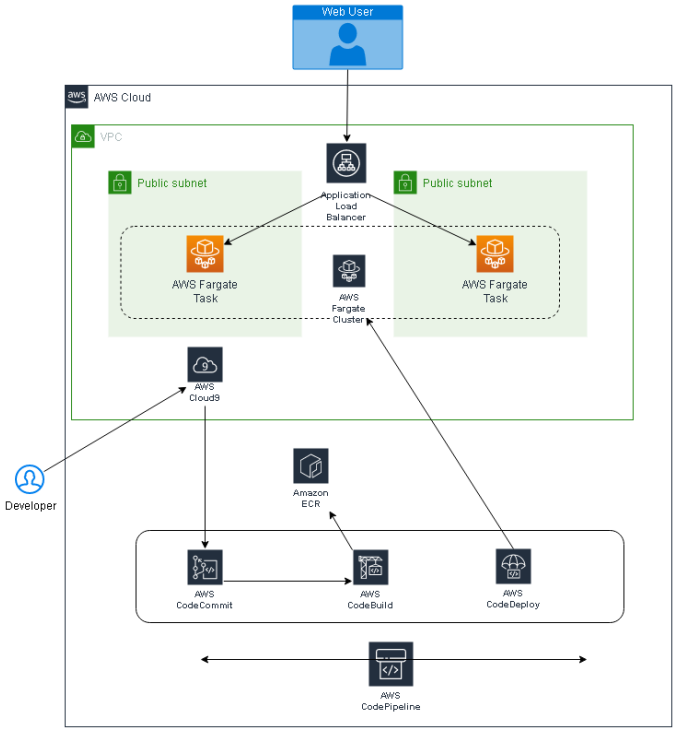
August 2022

Contents

Overview

This application walks you through creating a very simple PHP web application using the following AWS services:

- AWS Cloud9 – Development Environment
- AWS ECR – Container Registry
- AWS ECS Fargate – serverless container compute service, integrated with EC2 Load Balancing
- AWS CodePipeline – CI/CD orchestration with CodeCommit, CodeBuild and CodeDeploy



Creating and Deploying the App

This takes you through some quick examples and configuration for setting up an application lifecycle using the aforementioned AWS Services.

Configure and deploy a web app on ECS Fargate

For this example, we are going to use a very simple sample PHP web application. You can download the source code for this example from the following link:

https://github.com/jahoog/webphp-sample/blob/e53831c68346e8fc99a4f4a22c1d1e82ccf0dc96/archive/webphpSimple.zip?raw=true

Use Cloud9 to Create a Docker Container

The first thing we need to do is create a Docker Container. You can do this on your local workstation, but for this demo we are going to use a Cloud9 instance and create a docker image from there.

To setup a Cloud9 environment, you can follow the instructions here:
https://docs.aws.amazon.com/cloud9/latest/user-guide/create-environment.html

Once we login to our Cloud9 environment, we can use the terminal to download our sample php application and create a docker image. Use the following commands to do this.

```
mkdir webphp
cd webphp
curl -L -o webphp.zip https://github.com/jahoog/webphp-
sample/blob/e53831c68346e8fc99a4f4a22c1d1e82ccf0dc96/archive/webphpSimple.zip?raw=true
unzip webphp.zip
rm webphp.zip
```

```
hoog:~/environment/webphp $ curl -L -o webphp.zip https://github.com/jahoog/webp
96/archive/webphpSimple.zip?raw=true
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100   844  100   844    0     0  22546      0 --:--:-- --:--:-- --:--:-- 22546
hoog:~/environment/webphp $ unzip webphp.zip
Archive:  webphp.zip
  inflating: index.php
  inflating: styles/style.css
hoog:~/environment/webphp $ rm webphp.zip
```

Create the Docker file

A Docker file is a text document that contains the specs for the Docker image build. For more information you can view the Dockerfile reference page:
https://docs.docker.com/engine/reference/builder/

In this case, we are going to simply create a new Dockerfile:

```
touch Dockerfile
```

Then we will open the Dockerfile in our editor and enter the following into the Dockerfile and save the file:

```
FROM webdevops/php-apache-dev:latest

COPY . /app/
```

The above Dockerfile defines the layers within our docker image. We are going to have a simple 2 line file that does this:

Line 1: use a public php/apache based image as the base

Line 2: copies our application we unzipped into the /app/ folder of that image.

Build, Run and Test the docker container

Use the following commands to build, run and test the docker container:

```
docker build -t webphp .
docker images
```

Validate that the image was created:

```
[ec2-user@ip-172-31-9-18 webphp]$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
webphp                    latest              6478cfd2d763        24 minutes ago      563MB
```

Now let's run the container:

```
docker run -d -p 80:80 webphp
```

This runs the docker container in "detached" mode (specified by the –d option) and exposes the container's port 80 on the host's port 80 (specified by the –p option). We can view the running container with the following command:

```
docker ps
```

```
[ec2-user@ip-172-31-9-18 webphp]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
      PORTS                                  NAMES
f89441ae0456        webphp              "/entrypoint supervi…"   3 minutes ago       Up 3 minutes
      443/tcp, 0.0.0.0:80->80/tcp, 9000/tcp   kind_albattani
```

We can now test the application locally using curl:

```
curl http://localhost
```

```
[ec2-user@ip-172-31-9-18 webphp]$ curl http://localhost
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My test page</title>
    <link href='http://fonts.googleapis.com/css?family=Open+Sans' rel='styl
>
    <link href="styles/style.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <h1>My PHP Website #1</h1>
    <img src="http://php.net/images/logos/new-php-logo.png" alt="PHP Logo">

    <p>We are live!</p>

    <ul> <!-- changed to list in the tutorial -->
      <li>Time: Today is 2018/06/28</li>
    </ul>


  </body>
```

If you would like to stop your application, simply issue this command:

```
docker stop <container id or name>
```

```
</html>[ec2-user@ip-172-31-9-18 webphp]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  C
        PORTS                                      NAMES
f89441ae0456        webphp              "/entrypoint supervi…"   1
        443/tcp, 0.0.0.0:80->80/tcp, 9000/tcp   kind_albattani
[ec2-user@ip-172-31-9-18 webphp]$ docker stop kind_albattani
kind_albattani
```

Push your docker container to Amazon ECR

Amazon Elastic Container Registry gives you a private repository for your Docker containers. To push your container, you can follow the steps at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html#use-ecr

Note: for the following steps, you will need an IAM user that has access to push repositories to ECR (see https://docs.aws.amazon.com/codedeploy/latest/userguide/getting-started-create-iam-instance-profile.html)

```
aws ecr create-repository --repository-name webphp
```

```
[ec2-user@ip-172-31-9-18 webphp]$ aws ecr create-repository --repository-name webphp
{
    "repository": {
        "registryId": "396459200938",
        "repositoryName": "webphp",
        "repositoryArn": "arn:aws:ecr:us-east-1:396459200938:repository/webphp",
        "createdAt": 1530156048.0,
        "repositoryUri": "396459200938.dkr.ecr.us-east-1.amazonaws.com/webphp"
    }
}
```

```
docker tag webphp <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/webphp
```

```
[ec2-user@ip-172-31-9-18 webphp]$ docker images
REPOSITORY                                                 TAG                 IMAGE ID
TED             SIZE
396459200938.dkr.ecr.us-east-1.amazonaws.com/webphp        latest              6478cfd2d763
t an hour ago   563MB
```

```
$(aws ecr get-login --no-include-email)
```

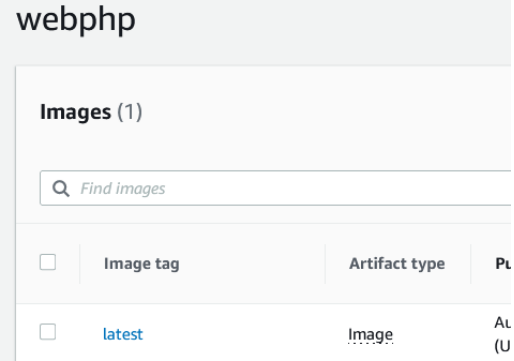
This will return a login command to ECR and run it automatically in our shell. You may get a warning, but as long as “Login Suceeded” is returned, you should be able to push the image to ECR.

```
[ec2-user@ip-172-31-9-18 webphp]$ $(aws ecr get-login --no-include-email)
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded
```

Use docker push command to push to ECR, using the repository URI which was returned in our create-repository step or is available through the ECR console.

```
docker push <aws_accountid>.dkr.ecr.us-east-1.amazonaws.com/webphp
```

Once complete, you should now see the first (and latest) image in the webphp repository in the ECR console:



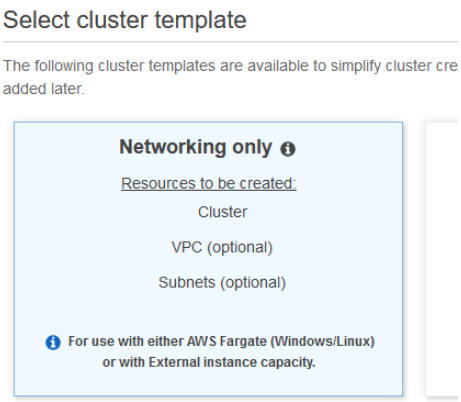
Deploying our image to ECS

To deploy to ECS, we will follow these 3 steps:

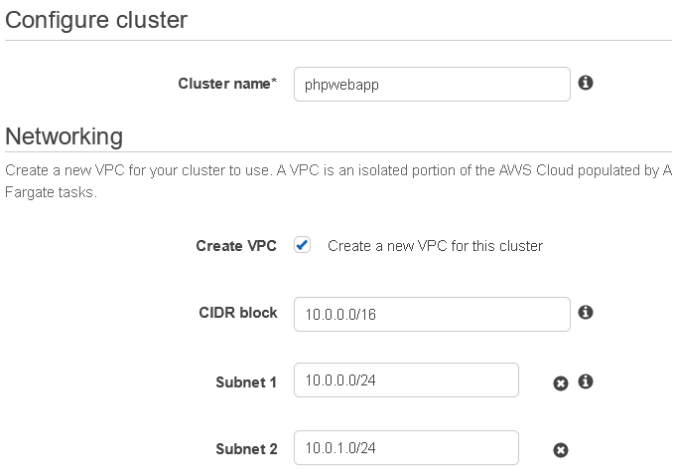
1. Create a Cluster
2. Configure a task definition
3. Configure a service

Creating a Cluster

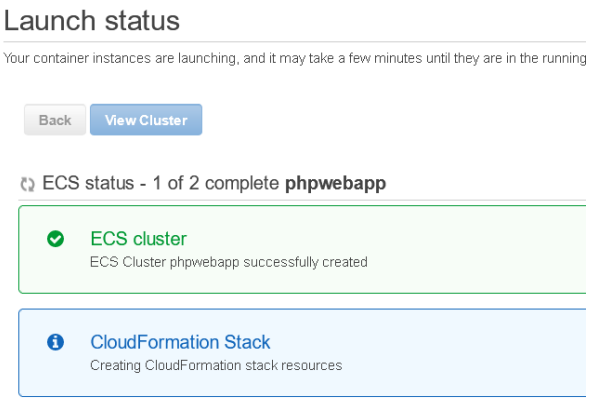
From the ECS Console, choose Create Cluster. We will use Networking Only, which means AWS Fargate will manage the underlying compute so that we can focus on running our application. Optionally, you can use EC2 Linux + Networking, which gives you complete control of the underlying EC2 instances but also adds additional overhead to manage.



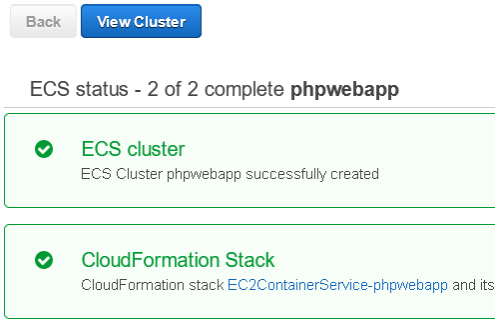
Enter a cluster name and then create a new VPC. You can leave the default networking configuration:



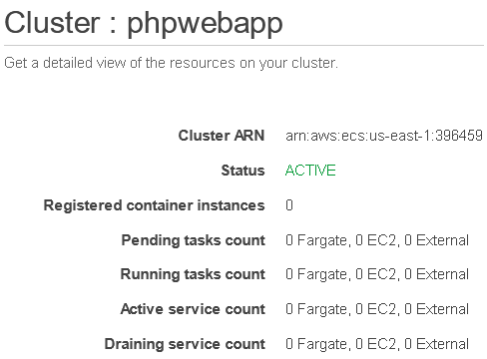
Click “Create” and the provisioning of your cluster resources will begin:



Once complete, you can click “View Cluster”



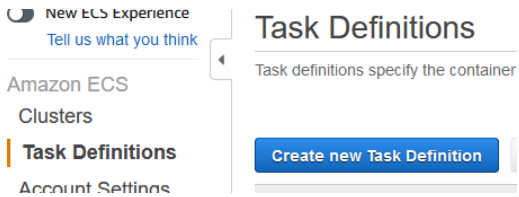
Once completed, you can view your cluster and ensure that it is active. No tasks will be running at this point:



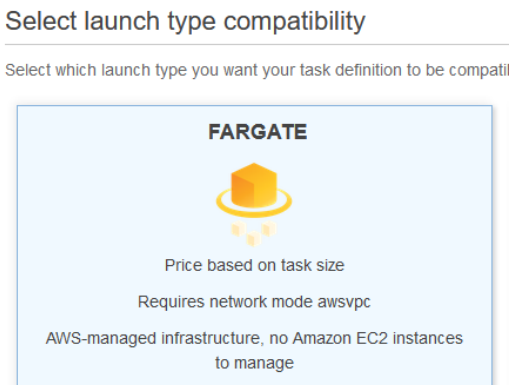
Create a Task Definition

You will need to create a task definition next. This describes the containers that will be deployed as tasks to your cluster.

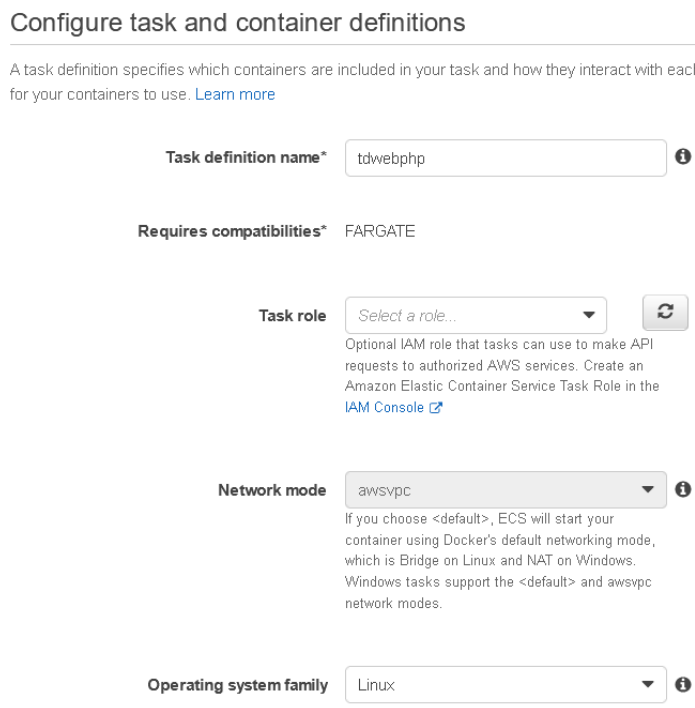
In the ECS Console, on the left menu choose Task Definitions and click “Create new task definition”.



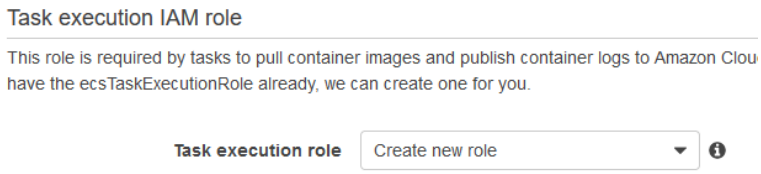
Select Fargate as the launch type compatibility (or match whatever cluster template you deployed).



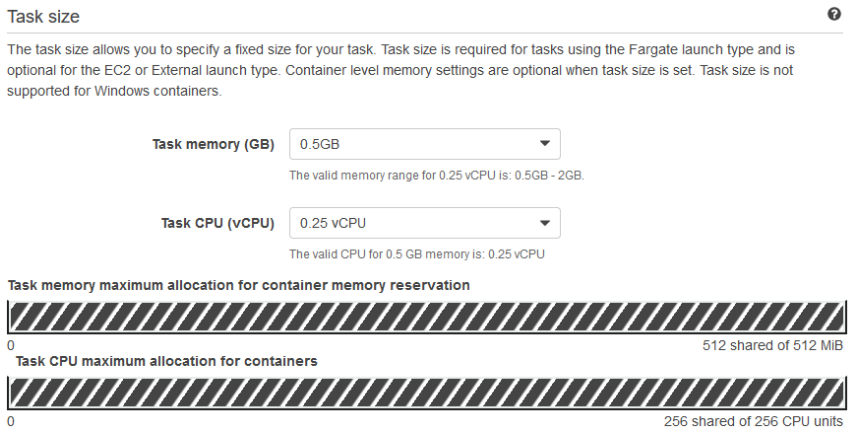
Click “Next Step”. Enter a Task Definition name and set the Operating System family to Linux.



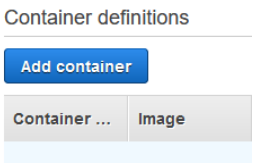
Choose “Create New role” for Task Execution IAM role (or use existing if you have one). This is used for logging metrics to Cloudwatch.



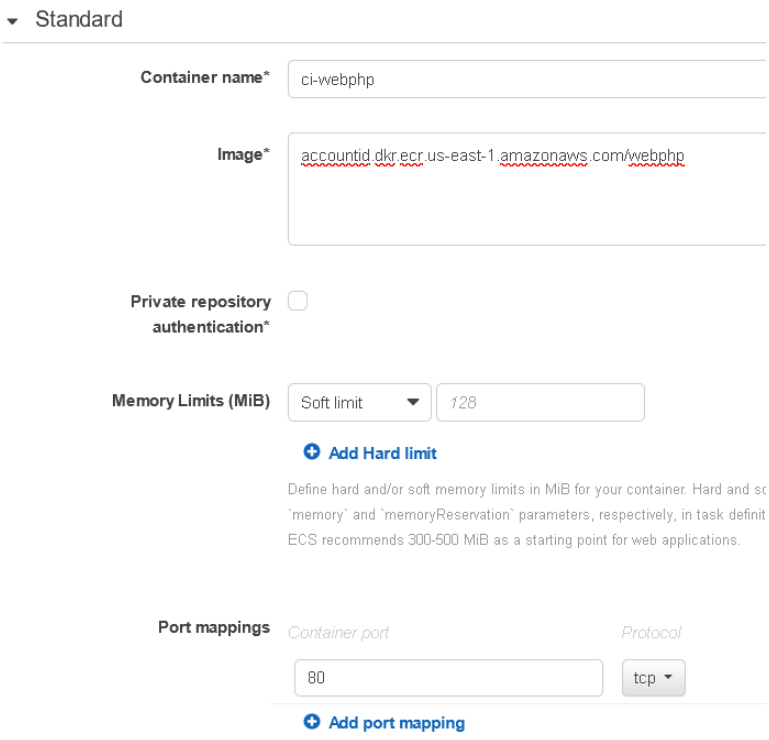
Under Task Size section, choose an acceptable task memory and cpu.



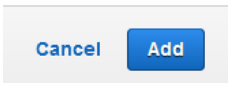
Next, click “Add Container”.



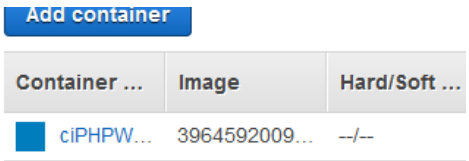
This will pop up a dialog box where we can enter information about the container image we will be hosting in our container tasks. Enter the container name and ECR image location, which was created earlier, and add the port that is exposed from the container.



Leave the additional sections as default settings and click “Add”.

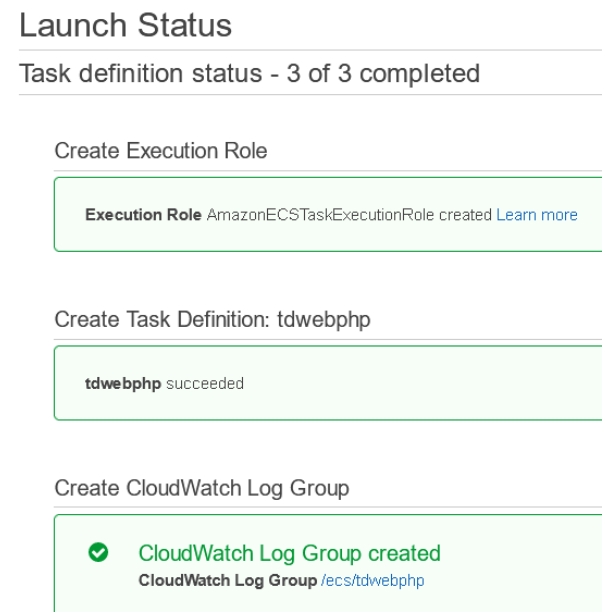


Verify that your container was added successfully.



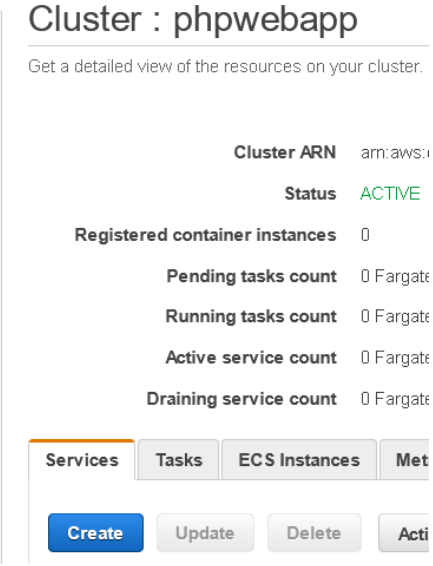
Click Create to finish creating your task definition.

Verify the task definition is launched successfully.



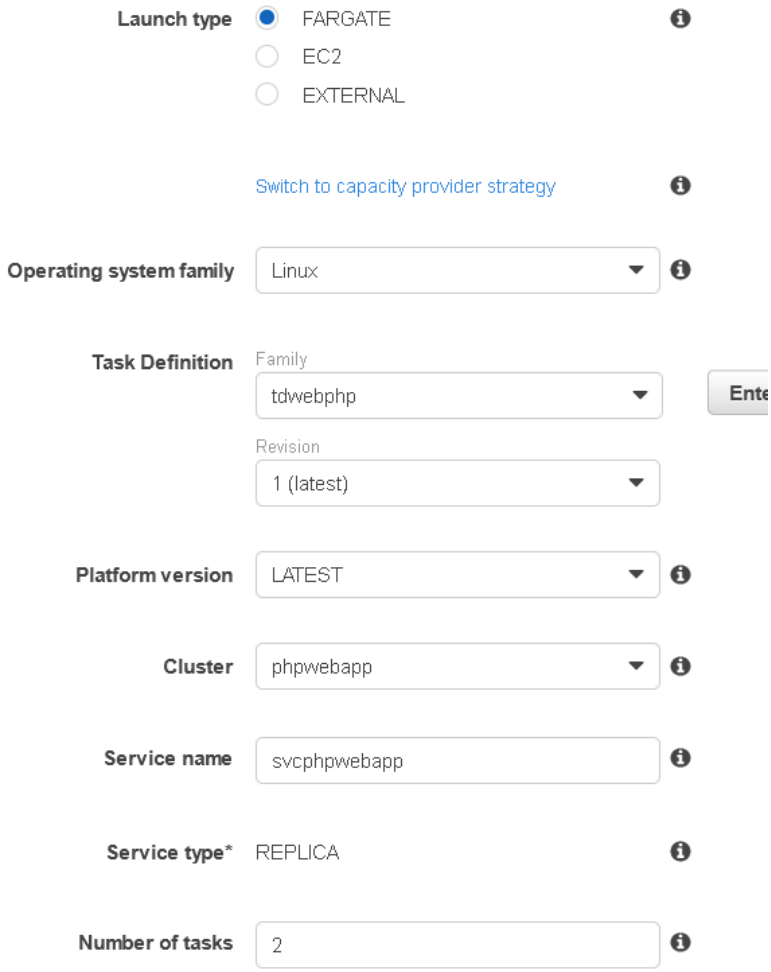
Create a Service

ECS Fargate containers can be deployed as single tasks, or they can include additional configurations like auto-scaling and load balancing via a service. To create service, go to the cluster we previously created and under the Services tab, click “Create”.



For the service properties, choose the following:

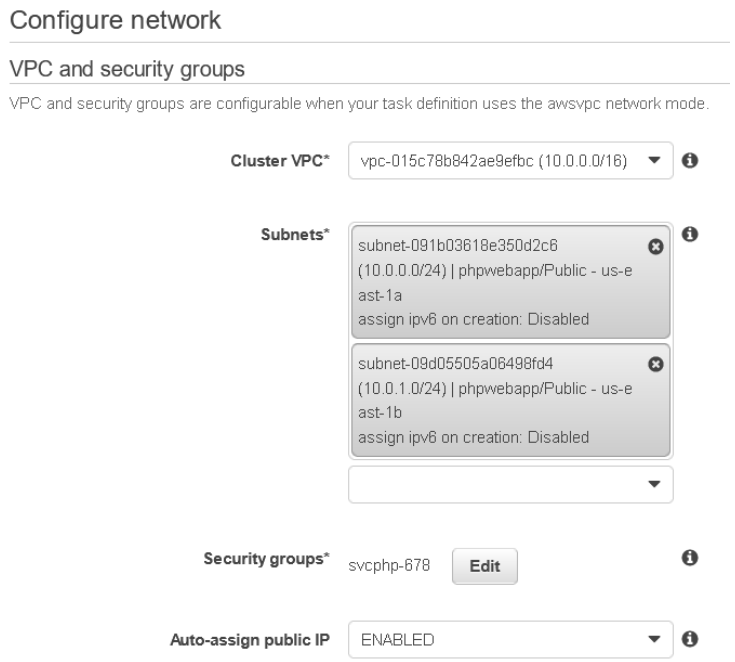
- Fargate as the launch type (or EC2 if appropriate)
- Linux as the Operating system family
- Select the task definition you created earlier from the drop down
- Provide a service a name
- Enter “2” as the number of tasks. This will distribute 2 tasks across our cluster



NOTE: Although you can certainly enter more than 2 tasks, ECS will not be able to deploy more than one task on each instance because we expose a static port 80 from our container. If we had specified ephemeral ports (using "0" instead of 80), then we could have multiple tasks of the same container running on each ECS instance.

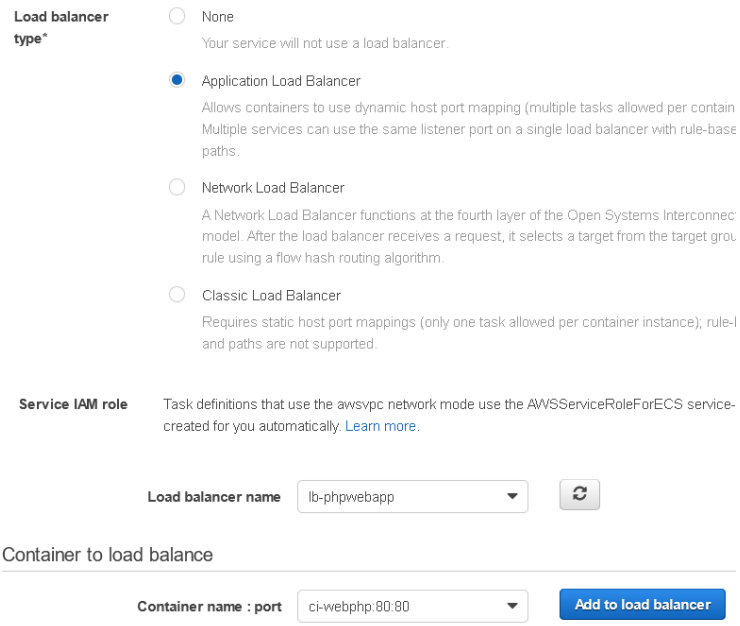
Keep "Rolling update" as the Deployment Type and click "Next Step".

On Step 2, choose the VPC and subnets that were created earlier in the Fargate cluster creation. You can keep the default security group and keep Auto-assign public IP to "ENABLED". (Note: for production environment, the Auto-assign public IP should be "DISABLED" and the tasks should be launched in a private subnet, but to do this there are some pre-requisites to ensure that the tasks can communication with necessary AWS services: https://aws.amazon.com/premiumsupport/knowledge-center/ecs-fargate-tasks-private-subnet/. So for now setting the property to ENABLED will do the trick.)

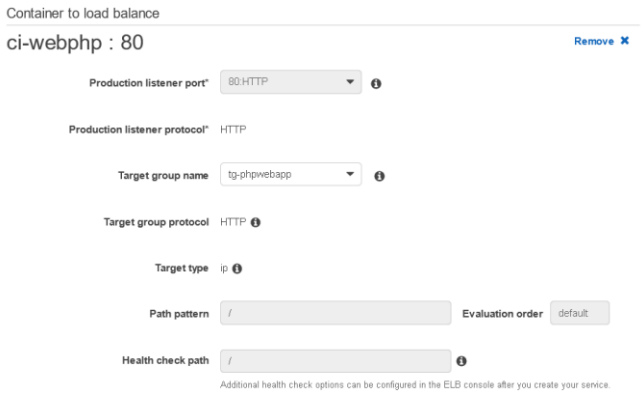


In the Load Balancing section, you can optionally configure a load balancer. Fargate does not automatically create the load balancer for you, so if you wish to configure this you will need to first create an Application Load Balancer (https://docs.aws.amazon.com/elasticloadbalancing/latest/application/create-application-load-balancer.html).

In our example, I am going to choose an empty Application Load Balancer that I created for this exercise. and then under Container to Load Balance, choose the one and only container available and click “Add to Load Balancer”:



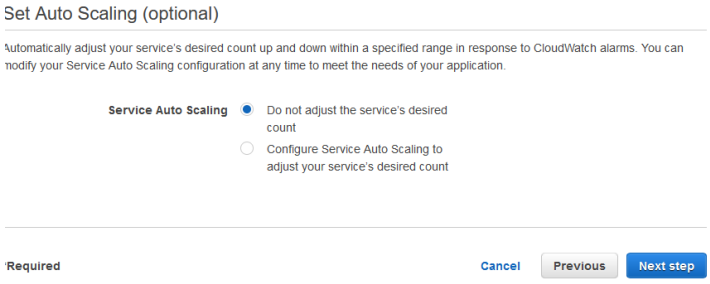
When you click “Add to Load Balancer”, choose the listener port of 80 and the target group with a target type of “ip” and protocol of “HTTP”. Optionally, you can have the Fargate service create these for you.



Once the ECS service launches new tasks, it will automatically register those tasks in the load balancer target group. Click “Next Step” to continue.

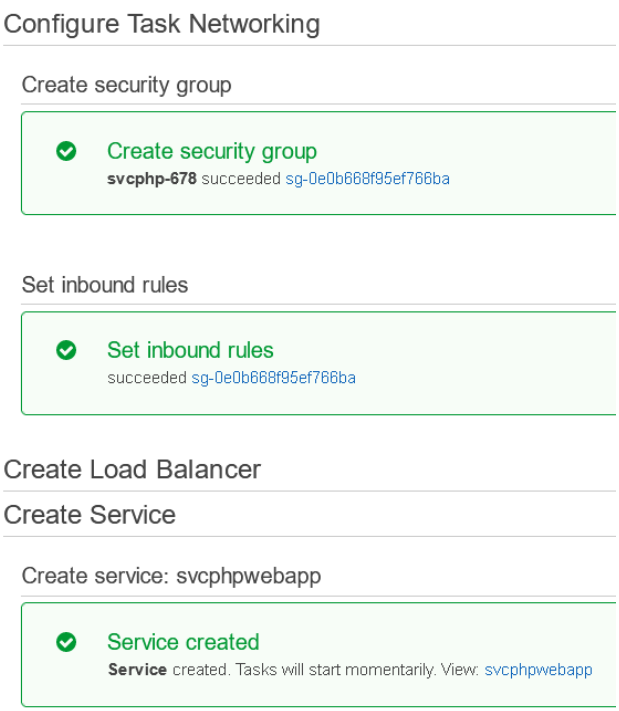
Under Set Auto Scaling, we are going to not adjust the size. You could optionally do this and the service will look at target metrics to scale the number of tasks in your service (https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-autoscaling-stepscaling.html).

Click “Next Step”.

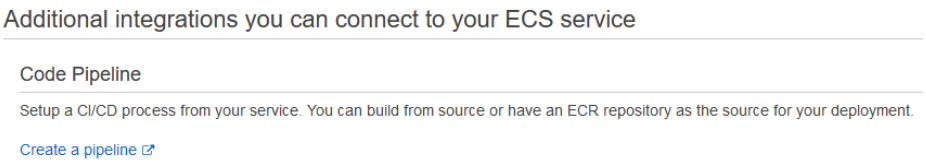


On the final screen, review your settings, and then click “Create Service”.

Ensure that your service was created successfully.

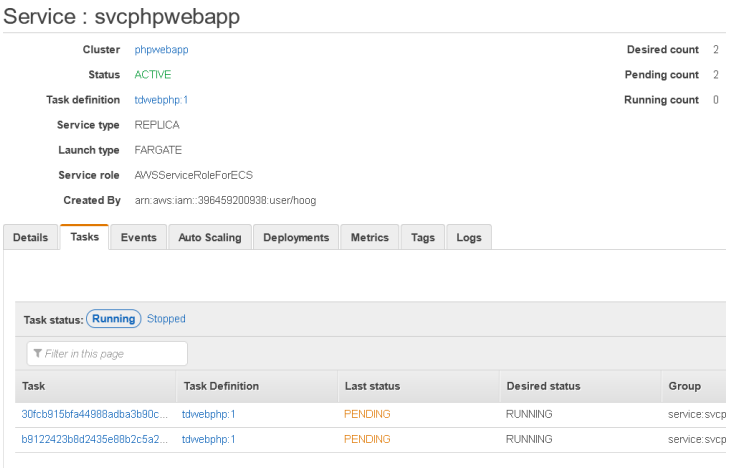


Notice that there is an option to create a Code Pipeline so that your app is automatically updated when a build is triggered. We will do this later.

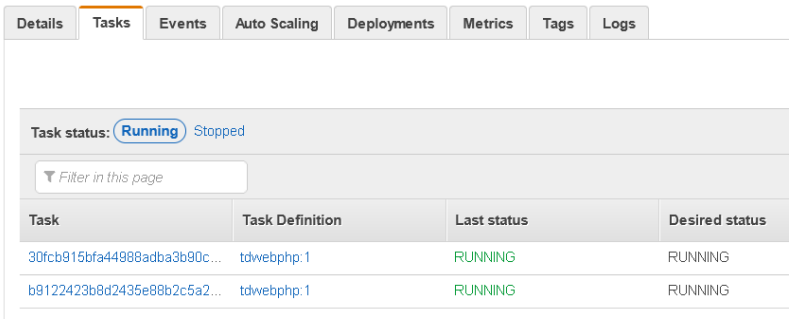


Click “View Service”.

This will take you to the tasks tab, where you will be able to see the tasks that your service deployed:



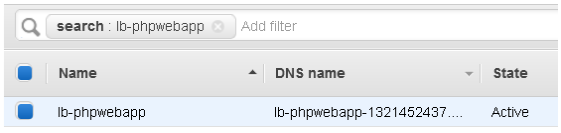
Initially, you will see the tasks in a pending state and then eventually they will switch to a RUNNING status.



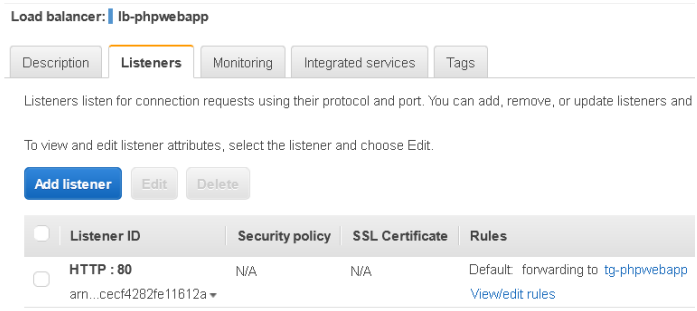
To test your deployment, you can first of all test directly to the ECS instance. This will only work if your security group has port 80 open from the instance to the internet. If so, you can click on one of the running tasks and in the detail section copy the public IP address under the Network section:

Go to a browser and navigate to http://<ip address>. This should load our very simple website:

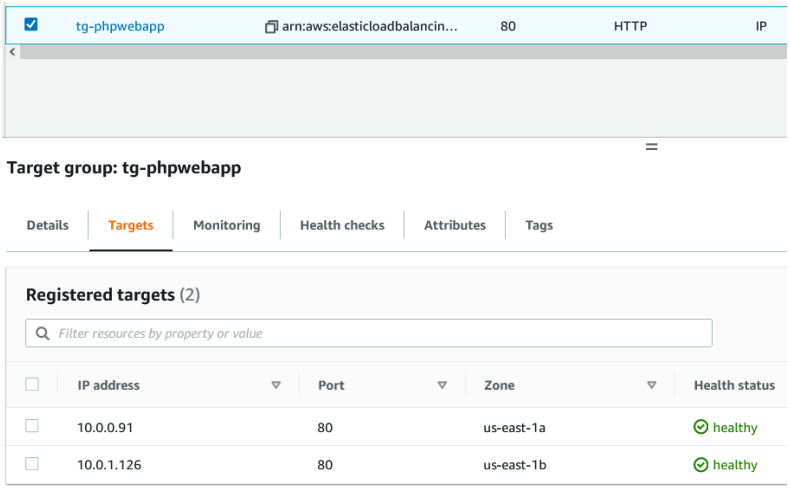
Now, a better production practice is to only expose the load balancer to the Internet as appropriate. Because we set up load balancing, you should be able to go to the EC2 Console, choose Load Balancers from the left menu, and find your Application Load Balancer in the list:



Click on the Listeners tab and in the Rules section it will have a forwarding rule. Click on the target group (e.g., tgFGPHPApp-Aug01):



When you click on the Target Group and then the Targets tab, you will be able to see the Fargate tasks that were added as targets to the group. Assuming security groups are open and the containers were deployed successfully, you should see 2 healthy targets:



Return to the load balancer and copy the DNS name, and then load it into a browser. You should see the website load correctly:

That completes deployment of our container into ECS Fargate.

Deploying our container application changes using CI/CD

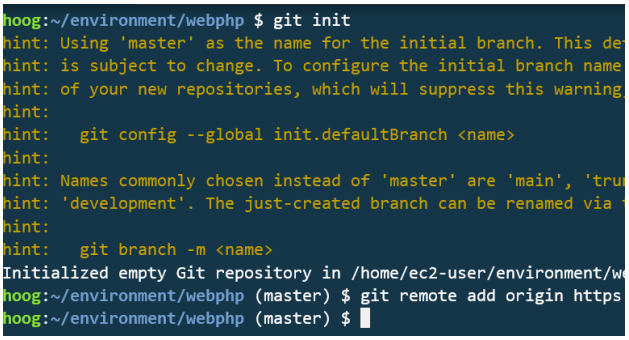
For this example, we are going to create a pipeline to orchestrate our continuous delivery (CI/CD) process. In our case, we are simply going to re-compile our container and push it into our ECS environment when changes are made.

Create a CodeCommit repository

The first thing we must do is deploy our code to a git based repo, in which case we are going to use AWS Code Commit.

Follow instructions to create a new repository and set up a connection. Return to Cloud9 and make sure you are in the webphp directory, and then initialize the webphp folder as a git repository:

```
git init
git remote add origin <repo url>
```



Once that is complete, in your development environment, add an additional file to our source folder called ***buildspec.yml.*** This will be used by CodeBuild to build our new container from the Dockerfile and source code. The contents of the buildspec.yml file should be as follows, but make sure to replace the repository name and container name with your account specific values:

```
version: 0.2

phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - export AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query "Account" --output=text)
      - aws --version
      - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login --username AWS --
password-stdin $AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com
      - REPOSITORY_URI=$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/<repository
name>
      - IMAGE_TAG=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - ls
      - docker build -t $REPOSITORY_URI:latest .
      - docker tag $REPOSITORY_URI:latest $REPOSITORY_URI:$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker images...
      - docker push $REPOSITORY_URI:latest
```

```
      - docker push $REPOSITORY_URI:$IMAGE_TAG
      - echo Writing image definitions file...
      - printf '[{"name":"<container name in task definition >","imageUri":"%s"}]'
$REPOSITORY_URI:$IMAGE_TAG > imagedefinitions.json
artifacts:
    files: imagedefinitions.json
```

The buildspec.yml file above is a standard docker build spec pulled from the Continuous Deployment with AWS CodePipeline Tutorial. You can also see more about the buildspec.yml reference at https://docs.aws.amazon.com/codebuild/latest/userguide/build-spec-ref.html

At this point you should have your local project folder with the sample PHP app, the Dockerfile, and the buildspec.yml file all as shown:

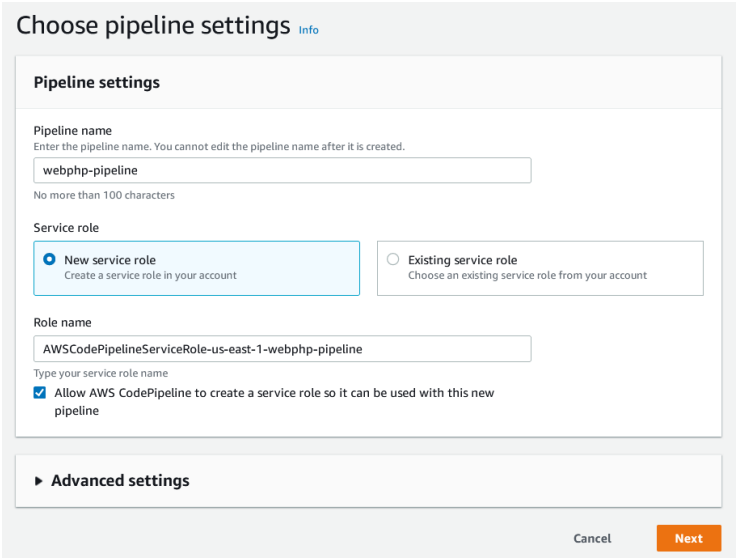
Now run the following git commands to upload to the CodeCommit repository:

```
git add .

git commit -m "Initial Commit"

git push
```

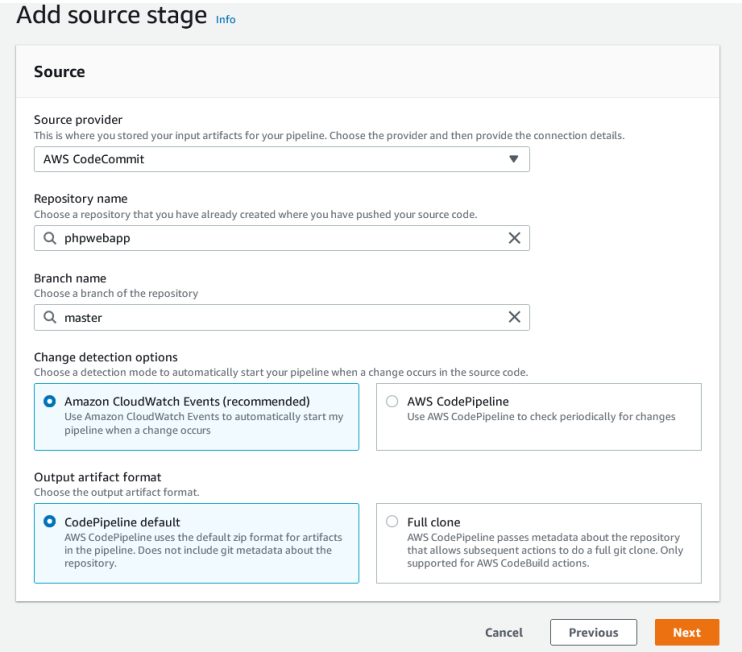
We now have our initial code committed to the git repo.

Create a CodePipeline

Now, let's create a new Code Pipeline. Give the Code Pipeline a name and use the default settings and click "Next".

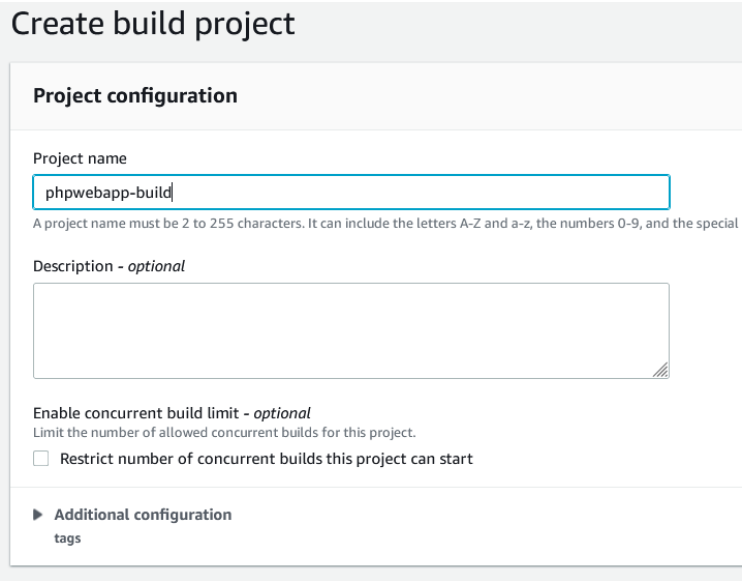


Select your source provider, repository name, and branch name:

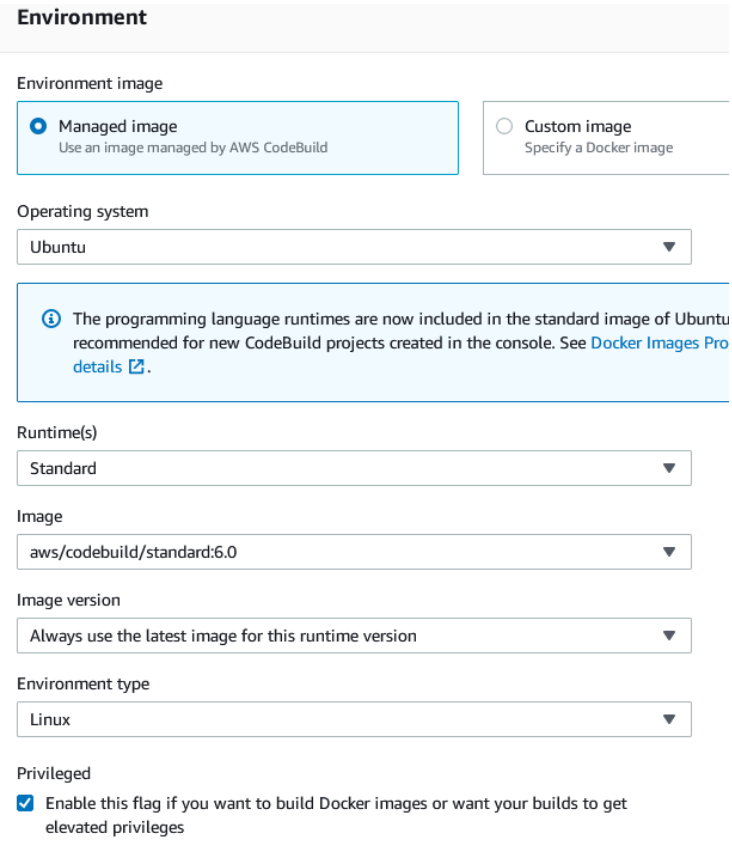


Click Next. Select CodeBuild as the build provider. This will be used to create our docker image and push to ECR. You will first need to select “Create Project”:

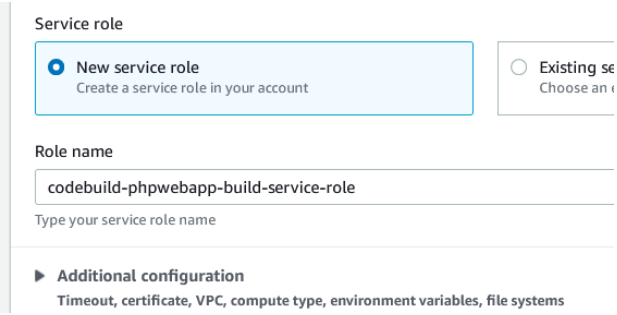
A popup window will appear. Enter a project name:



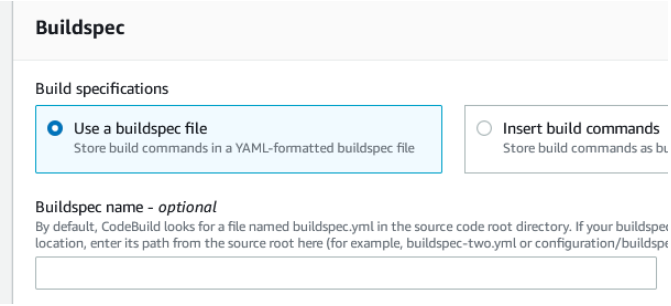
Next, under Managed Image, choose Ubuntu as the OS and use the Standard runtime. Check the box for privileged.



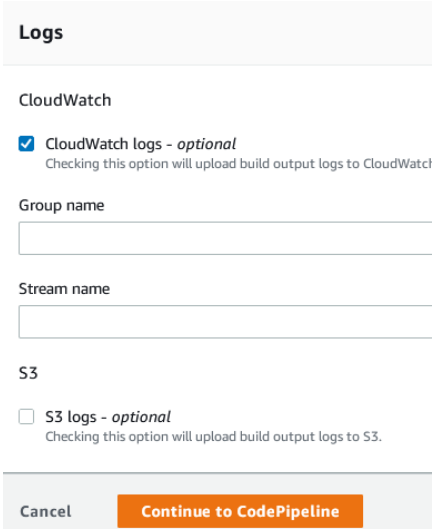
Leave the service role as the default:



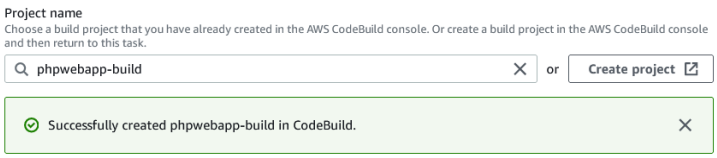
In the build specification, note that we will use the buildspec.yml file from the source code root directory. We added this earlier. No changes needed here:



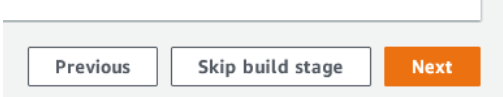
You may leave the final configurations for the CodeBuild project as default settings and choose Continue to CodePipeline to save the CodeBuild project.



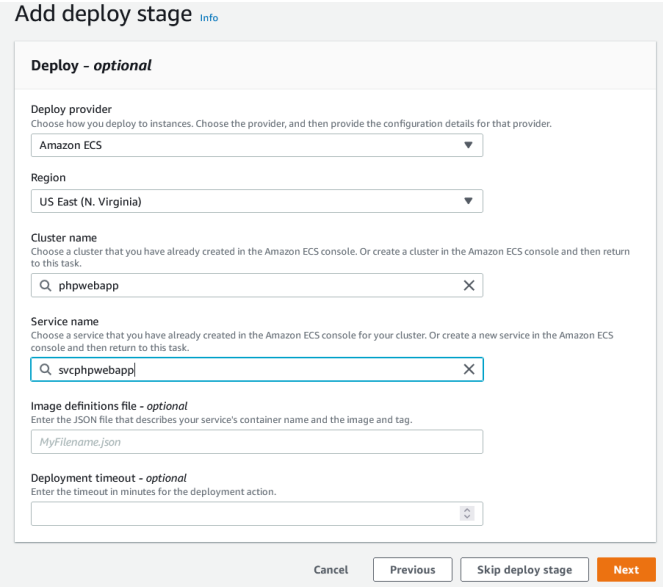
Once you do that, the popup window will close and you will get a success message.



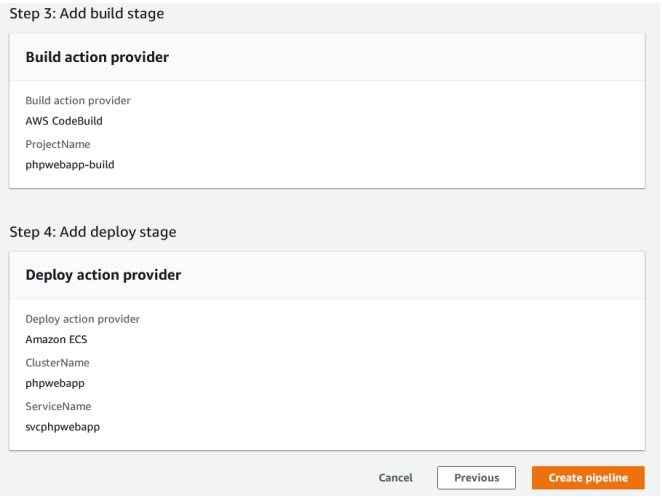
Click “Next”.



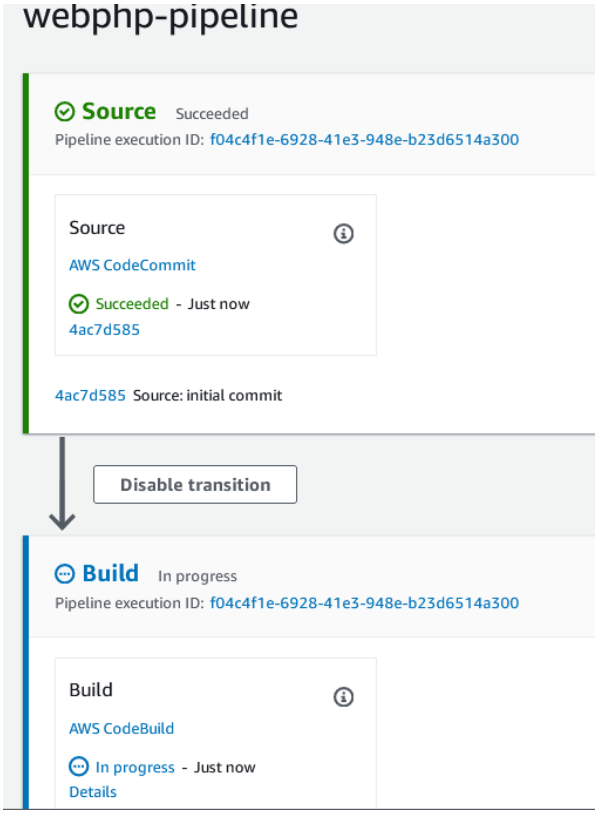
In this final step, we will deploy our code to our ECS Fargate Service by choosing Amazon ECS as the deployment provider, followed by the Cluster and service that we created earlier:



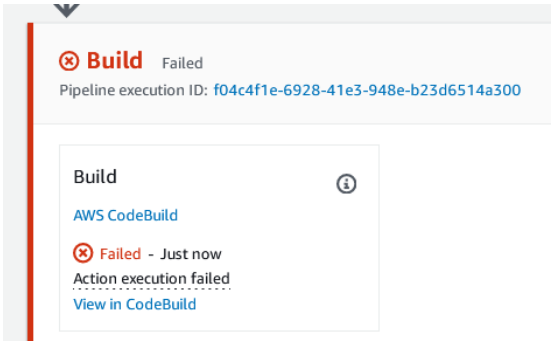
Click Next. Review our settings and click “Create Pipeline”.



We now have a Code Pipeline in place for deploying code changes to our ECS environment. It will automatically kick off when you save the Pipeline, triggering a CodeBuild from our Source Provider:



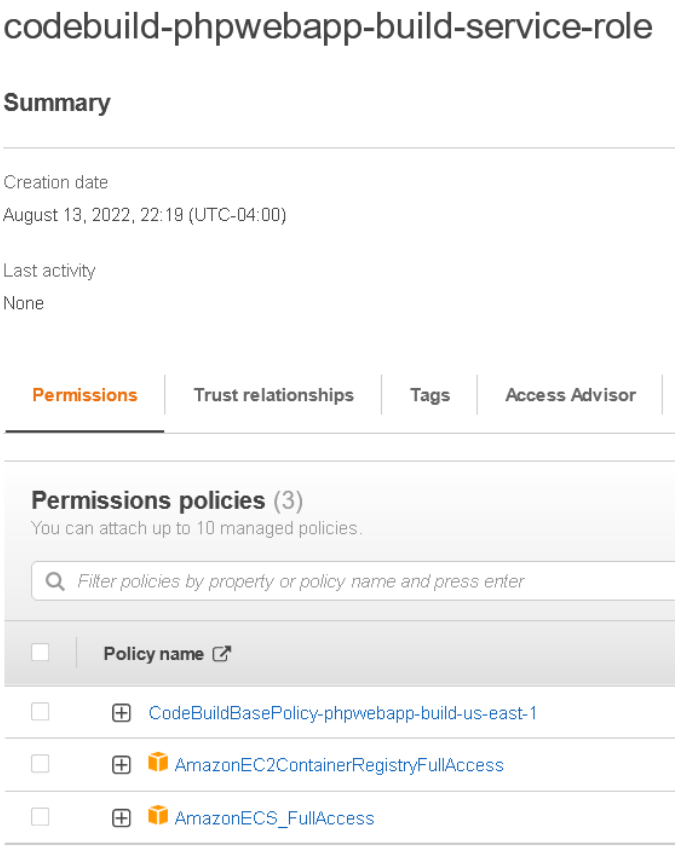
You may initially see an error in the Build stage.



If you do get an error, you can click on the CodeBuild link to see the error. It will probably look something like this:

```
An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn
c419-4829-8213-da56bf064862 is not authorized to perform: ecr:GetAuthorizationToken on resource: * be
Error: Cannot perform an interactive login from a non TTY device
```

This error indicates that a role does not have the necessary access to perform actions on ECR. To apply access, I will open the IAM Roles console and find the Role that was created earlier (codebuild-phpwebapp-build-service-role). Add 2 out of the box policies to give CodeBuild access to ECS and ECR (for production environments, you should reduce the permissions down to the exact actions needed):



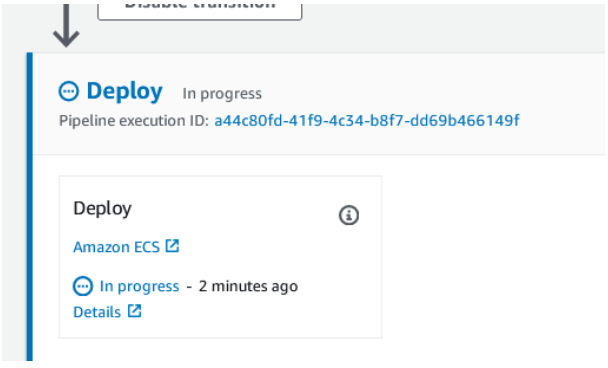
Once I save the role, you can kick off this pipeline by manually selecting the “Release change” button or you can push new changes to CodeCommit, which will automatically kick off the pipeline. Let’s go ahead and make a change by changing the header tag in the index.php file:

```
<h1>My Updated PHP Site</h1>
```

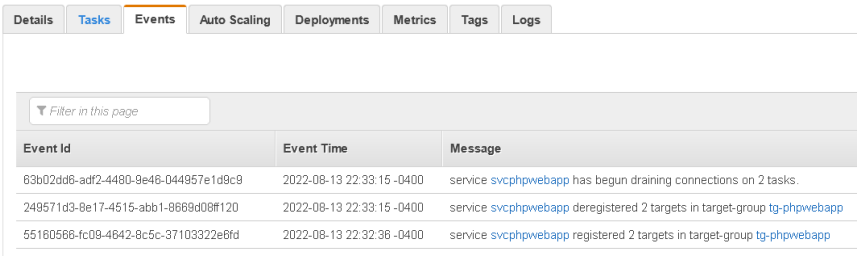
Update the repository:

```
git add .
git commit -m “Updated title”
git push
```

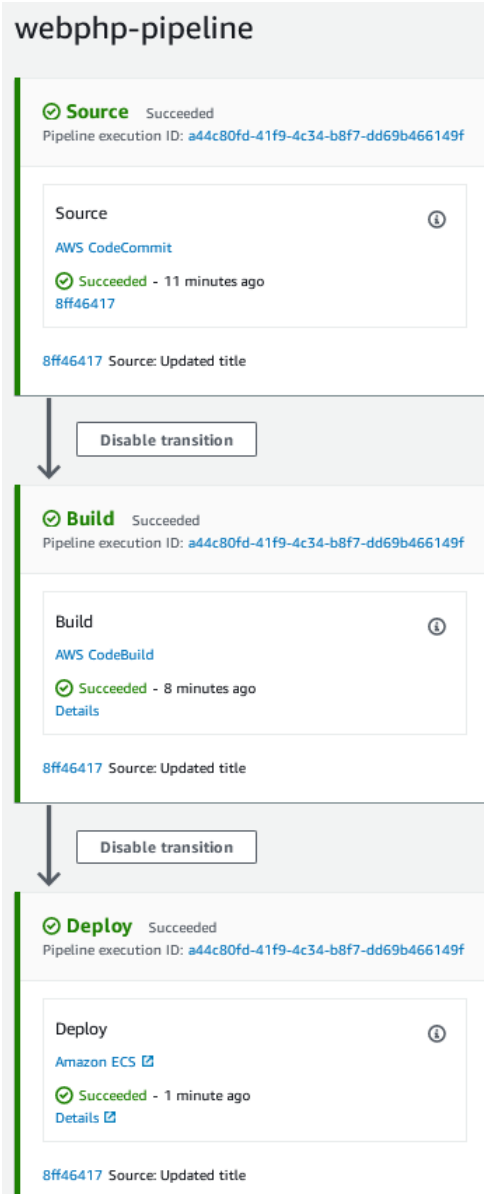
The pipeline will restart and take several minutes to build and push the new docker image to ECR and then deploy new tasks to ECS Fargate. You can click the Amazon ECS link to view the provisioning status in the cluster:



You will see in the Events tab the status of registering new task and de-registering old tasks. (This will take at least 5 minutes because the default connection draining for our load balancer is 300 seconds.)



Once all tasks in ECS have completed their provisioning/de-provisioning, all steps in the pipeline should show as “succeeded”.



You can now view your updated website in production by accessing the application load balancer of your front-end website. Great job!

Cleanup

You will want to manually delete resources that were created to avoid unnecessary charges. These include:

- CodePipeline and CodeBuild projects
- CodeCommit Repositories
- AWS Cloud9 Environments
- AWS EC2 Load Balancers, Target Groups and Security Groups
- AWS ECS Fargate Clusters
- AWS ECR Task Definitions
- AWS IAM Roles