AWS FARGATE SERVICES DEPLOYMENT WITH CONTINUOUS DELIVERY PIPELINE

AWS CodeBuild | AWS CodeCommit | CI/CD

beSharp | 5 April 2019

In this article, we explain how we have created a Continuous Delivery (CD) pipeline capable of producing a docker image for deployment on AWS ECS Fargate.

With the emergence of **AWS Fargate**, the realization of container-based services finally takes on a whole new meaning. In fact, before Fargate's release, the only way to use Amazon ECS was to provide a cluster of EC2 instances managed by Amazon (for software, updates, and configuration). This type of solution requires sustaining the costs of the clusters, plan oversizing to allow for the scaling of tasks, and lastly, configuring and maintaining a valid autoscaling system to avoid lacking adequate container resources.

AWS Fargate allows for all of this **management overhead to be handed to AWS**, i.e., to launch container-based services by paying only for the actual execution time. No need to worry about the underlying cluster—the focus can instead be placed on service development.

With AWS Fargate, AWS is making the container a top-tier object in computing solutions.

Automating the deployment of container-based services is fundamental to fully take advantage of AWS Fargate and AWS Cloud potential.

Here is our solution for implementing a CD pipeline that can put in production every push on the selected repository branch.

Key infrastructure services include:

- Amazon Elastic Container Service (Amazon ECS)
- Amazon Elastic Container Registry (ECR)
- Elastic Load Balancing

- (Optional) CodeCommit for the GIT repository. Any other CodePipeline-supported repository will be fine.

Let us start with a short glossary:

Amazon Elastic Container Service (Amazon ECS) is a container orchestration service. It supports Docker and allows for **easily running and resizing applications**. AWS Fargate facilitates starting and orchestrating container-based services by fully using AWS-managed clusters and paying on a container basis.

Amazon Elastic Container Registry (ECR) is a **fully Docker-managed image registry** that makes it easy for developers to store, manage, and distribute Docker container images.

It is possible to utilize the Elastic Load Balancing service to sort traffic via containers.

AWS Elastic Load Balancing automatically routes incoming application traffic between multiple destinations, including EC2, containers, IP addresses, and Lambda functions.

Elastic Load Balancing offers **three types of load balancing systems:**

1. Application Load Balancer
2. Network Load Balancer
3. Classic Load Balancer

The Application Load Balancer (ALB) is the load balancer for the services released on Fargate.

The Application Load Balancer systems are suitable for **balancing HTTP and HTTPS traffic**. They offer advanced request routing for the distribution of modern architectures, e.g. in microservices and containers. These systems operate at the level of individual requests (level 7) and route traffic based on the content of the request.

Without further delay, let us move on to the tutorial for creating a fully automated release pipeline.

Throughout the rest of the article, we will assume that the entire project code is in a **CodePipeline-compatible repository.**

Part 1: Preparation of the first docker image

First, prepare an image of our service for testing it both locally and on AWS.

It is, therefore, necessary **to add a Dockerfile to the project**, which will then be published in the repository. The file must contain instructions for building a container for all the software, dependencies, libraries, and configurations, as well as the package with our service.

This container can be safely tested locally or in a controlled environment in order to verify proper functioning.

Once the local tests are satisfying, one can proceed with the creation of an image and its publication on Amazon ECR.

The **creation of an ECR repository** follows, and the only data it requires is a valid name. Our docker image can then be uploaded to ECR by simply following the login and push instructions.

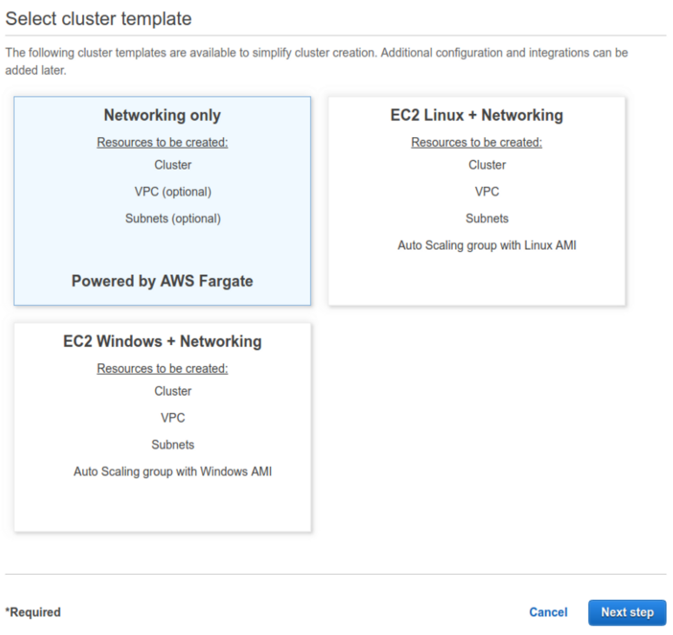
```
$(aws ecr get-login --no-include-email --region <regione>)
docker build -t <nome immagine> .
docker tag <nome immagine>:latest <ecr url>:latest
docker push <ecr url>:latest
```

Part 2: Configuration of ECS Fargate and Networking

Our service requires a **Load Balancer** to route traffic between replica containers.

For this reason, we need to **create an Application Load Balancer** whose configuration can be left blank. Defining behavioral details of the ALB is unnecessary because ECS is going to manage it dynamically during the containers' scaling operations.

As for ECS, the first thing to do is to create a cluster. **Clusters** are nothing more than objects used to logically group services.


Select cluster template

The following cluster templates are available to simplify cluster creation. Additional configuration and integrations can be added later.

Networking only
Resources to be created:
Cluster
VPC (optional)
Subnets (optional)
Powered by AWS Fargate

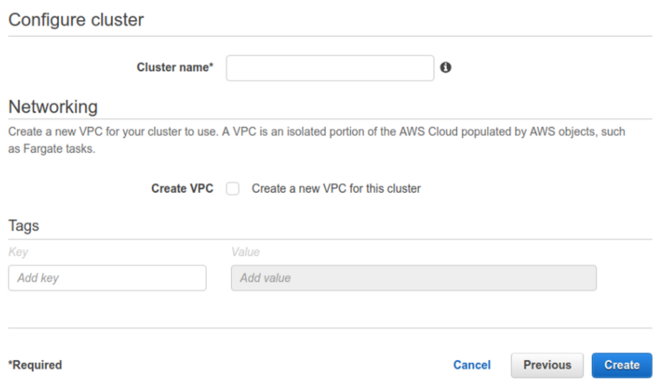
EC2 Linux + Networking
Resources to be created:
Cluster
VPC
Subnets
Auto Scaling group with Linux AMI

EC2 Windows + Networking
Resources to be created:
Cluster
VPC
Subnets
Auto Scaling group with Windows AMI

Access the ECS dashboard and select "create cluster".

From the wizard, choose "Networking only". This configuration tells AWS to use AWS Fargate for this virtual cluster.

If desired, select the name and a new VPC in the second and last wizard step. Otherwise, use one that has already been configured on your account.


Configure cluster

Cluster name*

Networking

Create a new VPC for your cluster to use. A VPC is an isolated portion of the AWS Cloud populated by AWS objects, such as Fargate tasks.

Create VPC ☐ Create a new VPC for this cluster

Tags

Key	Value
Add key	Add value

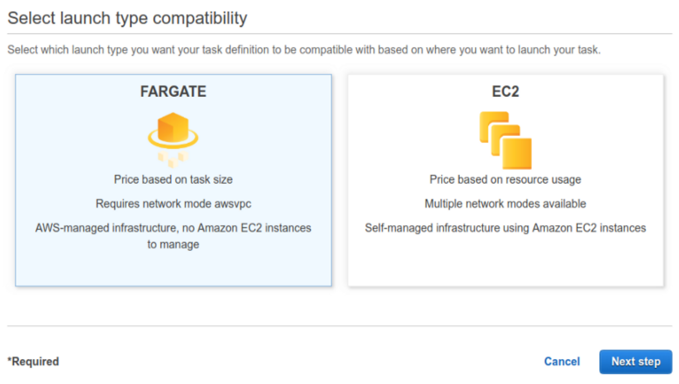
*Required

The second step **creates a task definition.** This object collects information about the task, i.e., name, description, IAM roles for deployment and execution, the size of the task in terms of RAM and CPU, and the specifications of the container that will host it.

Select the previously saved docker image on ECR to configure the container.

Simply select "Create task definition" from the appropriate screen in the ECS area.

It is essential to choose AWS Fargate as the wizard's first step. Then input the requested data following the instructions and provide adequate sizing for the task.


Select launch type compatibility

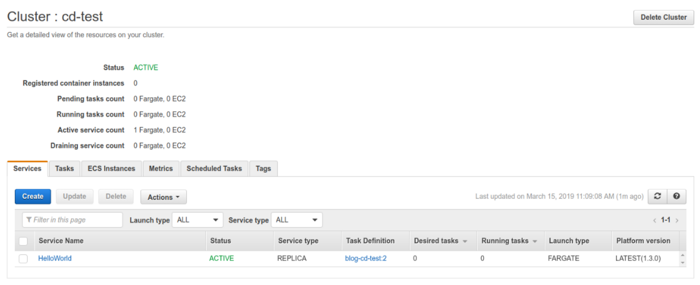
Select which launch type you want your task definition to be compatible with based on where you want to launch your task.

FARGATE	EC2
Price based on task size	Price based on resource usage
Requires network mode awsvpc	Multiple network modes available
AWS-managed infrastructure, no Amazon EC2 instances to manage	Self-managed infrastructure using Amazon EC2 instances

*Required

The last object to be configured is called service (Service).

A service is defined by a task and a set of parameters that specify how many instances of the task are required as a minimum, current, and maximum value to allow the service to function correctly.


Cluster : cd-test

Get a detailed view of the resources on your cluster.

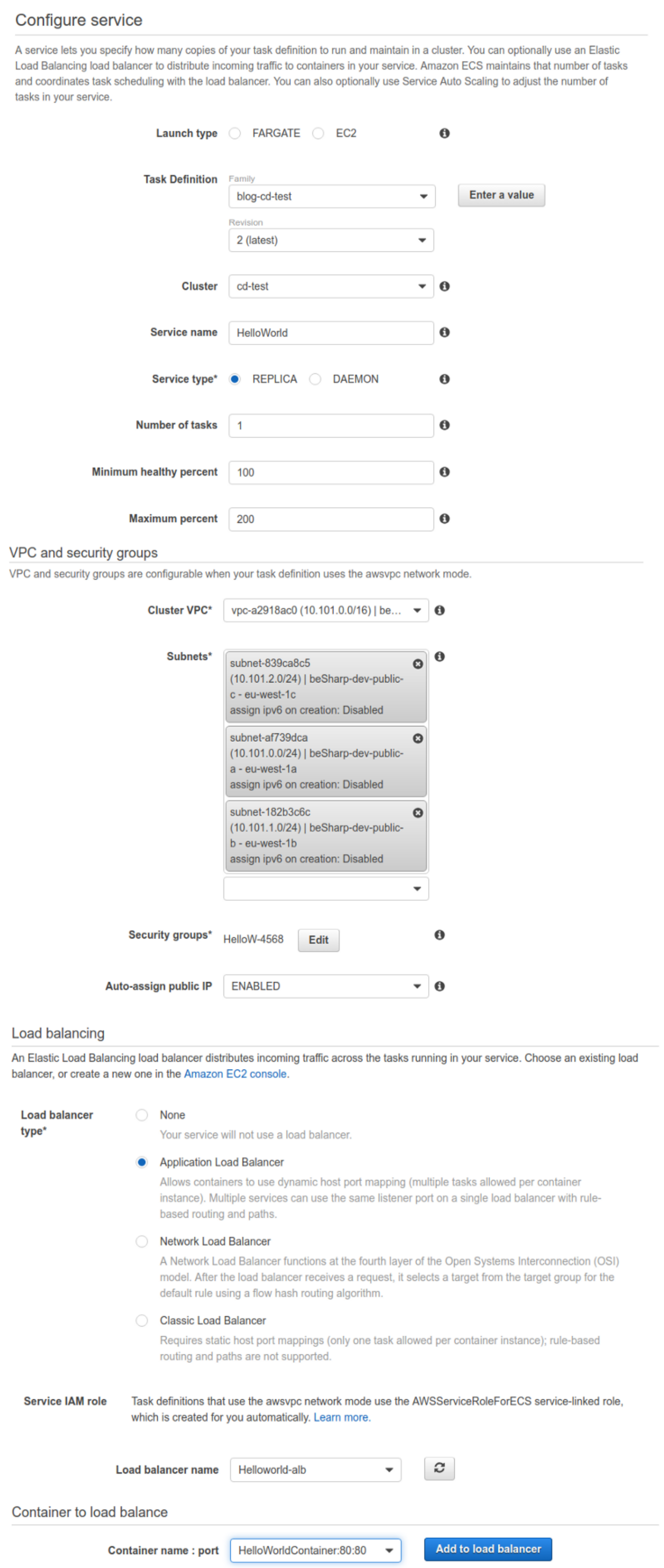
Status	ACTIVE
Registered container instances	0
Pending tasks count	0 Fargate, 0 EC2
Running tasks count	0 Fargate, 0 EC2
Active service count	1 Fargate, 0 EC2
Draining service count	0 Fargate, 0 EC2

Last updated on March 15, 2019 11:09:08 AM (1m ago)

Filter in this page Launch type ALL Service type ALL 1-1

Service Name	Status	Service type	Task Definition	Desired tasks	Running tasks	Launch type	Platform version
HelloWorld	ACTIVE	REPLICA	blog-cd-test:2	0	0	FARGATE	LATEST(1.3.0)

The creation procedure is no different from other configured objects.



Configure service

A service lets you specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS maintains that number of tasks and coordinates task scheduling with the load balancer. You can also optionally use Service Auto Scaling to adjust the number of tasks in your service.

Launch type	FARGATE EC2
Task Definition	Family: blog-cd-test (Enter a value)
	Revision: 2 (latest)
Cluster	cd-test
Service name	HelloWorld
Service type*	[✓ REPLICA] DAEMON
Number of tasks	1
Minimum healthy percent	100
Maximum percent	200

VPC and security groups

VPC and security groups are configurable when your task definition uses the awsvpc network mode.

Cluster VPC*	vpc-a2918ac0 (10.101.0.0/16) | be...
Subnets*	subnet-839ca8c5 (10.101.2.0/24) | beSharp-dev-public-c - eu-west-1c assign ipv6 on creation: Disabled
	subnet-af739dca (10.101.0.0/24) | beSharp-dev-public-a - eu-west-1a assign ipv6 on creation: Disabled
	subnet-182b3c6c (10.101.1.0/24) | beSharp-dev-public-b - eu-west-1b assign ipv6 on creation: Disabled
Security groups*	HelloW-4568 (Edit)
Auto-assign public IP	ENABLED

Load balancing

An Elastic Load Balancing load balancer distributes incoming traffic across the tasks running in your service. Choose an existing load balancer, or create a new one in the Amazon EC2 console.

Load balancer type*

- None
  Your service will not use a load balancer.
- [✓ Application Load Balancer]
  Allows containers to use dynamic host port mapping (multiple tasks allowed per container instance). Multiple services can use the same listener port on a single load balancer with rule-based routing and paths.
- Network Load Balancer
  A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it selects a target from the target group for the default rule using a flow hash routing algorithm.
- Classic Load Balancer
  Requires static host port mappings (only one task allowed per container instance); rule-based routing and paths are not supported.

Service IAM role: Task definitions that use the awsvpc network mode use the AWSServiceRoleForECS service-linked role, which is created for you automatically. Learn more.

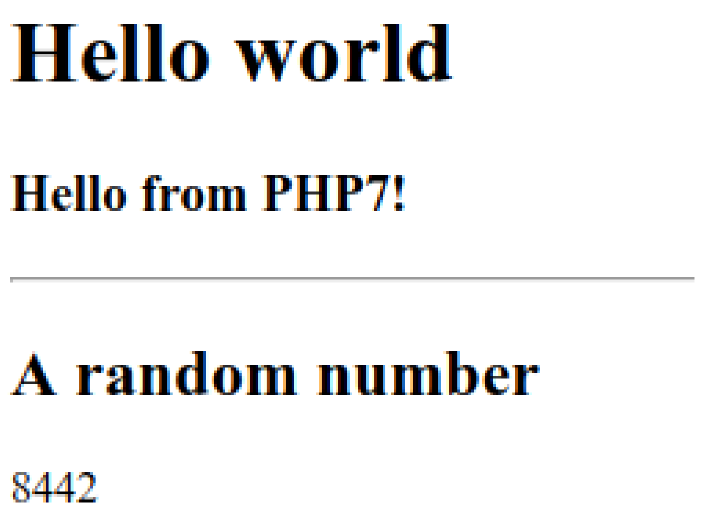
Load balancer name: Helloworld-alb

Container to load balance

Container name : port: HelloWorldContainer:80:80 (Add to load balancer)

Be attentive when selecting the VPC, subnets, and previously created load balancer.

The service should be visible by pointing the browser to the ALB URL at the end of the configuration.


Hello world

Hello from PHP7!

A random number

8442

Part 3: Automating the deployment

Once the entire environment has been manually configured, it is possible to **create and configure a pipeline that automatically deploys each code change.**

The file named **buildspec.yml** needs to be added to the repository root before starting the pipeline configuration. The purpose of the file is to contain the instructions for building a new image of our service.

In fact, we want to automate what was previously performed by hand, i.e., the construction of the docker image from the dockerfile and the code, its uploading to ECR, and lastly, ECS updating to perform the service deployment by using the new image.

Here is a sample version of the file to be added (buildspec.yml):

```
version: 0.2
```

```
phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - aws --version
      - $(aws ecr get-login --region $AWS_DEFAULT_REGION --no-include-email)
      - REPOSITORY_URI=<REPLACE THIS TEXT WITH THE URL OF THE IMAGE USED ON ECR>
      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
      - IMAGE_TAG=${COMMIT_HASH:=latest}
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - 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":"<replace this text with the container name used in task definition>","i
mageUri":"%s"}]' $REPOSITORY_URI:$IMAGE_TAG > imagedefinitions.json
artifacts:
    files: imagedefinitions.json
```

The parts to be edited with the specific names of your project are in bold.

CodePipeline is the AWS service for implementing the automation with very low management and configuration effort. We will rely on CodeBuild to execute the file with the instructions (buildspec.yml) to perform image build operations.

So let us start with **the pipeline configuration** by creating a new one on CodePipeline:

1. Select “Create pipeline”.
2. Follow the wizard. It will be possible to update your service automatically without any additional effort.
3. Choose your GIT repository (CodeCommit or compatible) as a source in the second step.
4. Next, configure the Build step. Select CodeBuild and the option for creating a new one.
5. Give a univocal name to the build project in the sub-wizard.
   Choose **Ubuntu** as the operative system and **Docker** as Runtime.
6. Then select the docker version that best suits your purposes.
7. Save by completing the last step of the sub-wizard and return to editing the pipeline.
8. Configure the Deploy step where you can choose **Amazon ECS**.
9. At step 4, indicate the cluster name previously configured by hand and the name of the service to be updated.
10. For step 5, simply select the option so that CodePipeline can create a role for the execution of the pipeline and configure it for us. This role is used by CodePipeline to modify ECS. It is managed by Amazon and has the minimum set of permissions to make everything work.
11. Save the pipeline.

At this point, the pipeline will try to run automatically, and it will *fail.*

This is an expected failure: the reason lies in the fact that the Wizard has created a CodeBuild role for us. However, this does not have all the necessary permissions to push the image to ECR.

To solve this, identify the generated role whose name follows this convention: code-build-build-project-name-service-role. Then add the following permissions:

AmazonEC2ContainerRegistryPowerUser to see the running pipeline.

If everything worked as expected, then the pipeline will now function. Further, the service will be automatically updated at every commit on the chosen branch.

Containers are increasingly at the center of the DevOps scene. As a result, it is important to know the available tools for making thoughtful and effective project choices. We hope to have been helpful to you in this regard.

Please share with us your results, observations, doubts, and ideas… Our team is looking forward to furthering this topic with you!

[ATTENTION, SPOILER!]

If you are intrigued by this subject, keep following us: ***a creative way to get a highly personalized automatic pipeline that uses containers as a means of automation is coming your way.***

Stay tuned! 😉

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all’avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it