AWS Project : Container Deployment with CICD

Documentation

By : Abouchiba Mohamed Yassir

To see the files used in this project go to this link:
https://github.com/M-Yassir/AWSProjects/tree/main/Container_deployment_with_CICD

Introduction

Project Description

This project demonstrates the development of a fully automated CI/CD pipeline for deploying containerized applications using Amazon Web Services (AWS). The system is designed to streamline the process of building, testing, and deploying applications, ensuring efficient and reliable delivery to end users.

The project is highly beneficial for organizations aiming to adopt modern DevOps practices, as it significantly reduces manual intervention, minimizes deployment errors, and accelerates the delivery of new features and updates. By leveraging AWS services such as CodePipeline, CodeBuild, ECR, ECS, and ALB, the pipeline ensures a seamless, scalable, and cost-effective deployment process. This approach not only enhances operational efficiency but also improves the overall reliability and performance of the application, providing a better experience for end users.

Through this project, I gained hands-on experience in CI/CD automation, containerization, cloud infrastructure, and AWS service integration, making it a valuable exploration of modern DevOps practices and scalable application deployment.

Key Features

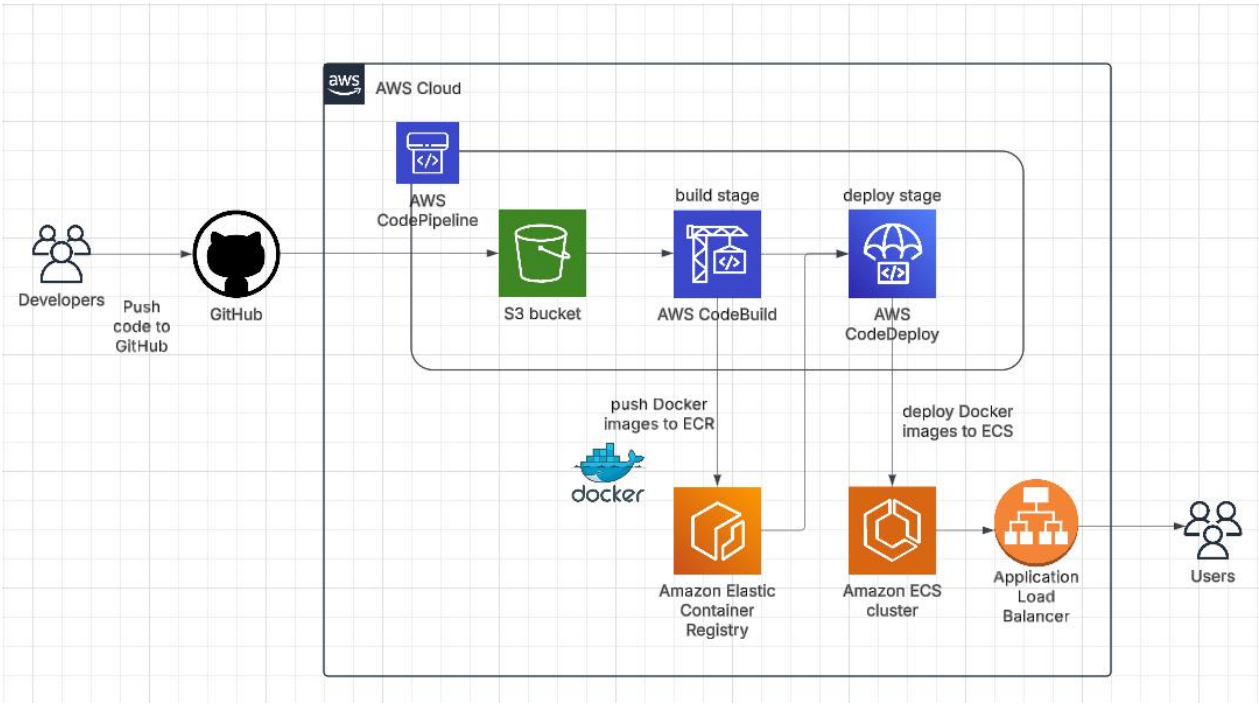
-Automated CI/CD Pipeline: The pipeline is orchestrated using AWS CodePipeline, which integrates with GitHub to detect code changes, triggers CodeBuild for Docker image creation, and uses CodeDeploy to update the ECS service. This ensures a fully automated and efficient deployment process.

-Docker Image Management: AWS CodeBuild pulls the latest code from GitHub, builds the Docker image using the provided Dockerfile, and pushes it to Amazon ECR. ECS then pulls the image from ECR to run the application in a scalable and managed environment.

-Scalable Application Deployment: The application runs on Amazon ECS, which manages the deployment of Docker containers using either EC2 instances or AWS Fargate. This ensures high availability, scalability, and efficient resource utilization.

-Traffic Routing with ALB: The Application Load Balancer (ALB) routes incoming traffic to the ECS tasks, ensuring even distribution and high availability for end users.

AWS architecture diagram



- Application workflow:

This application automates the CI/CD process for a Node.js app using AWS services and GitHub. The workflow begins when developers push code changes, such as the Node.js application code, Dockerfile, or configuration files, to the GitHub repository. This triggers AWS CodePipeline, which orchestrates the pipeline. In the Source Stage, CodePipeline pulls the latest code from GitHub, preparing it for the build process.

Next, in the Build Stage, AWS CodeBuild compiles the Node.js app and builds a Docker image using the provided Dockerfile. Once the image is created, it is pushed to Amazon Elastic Container Registry (ECR) for secure storage and versioning.

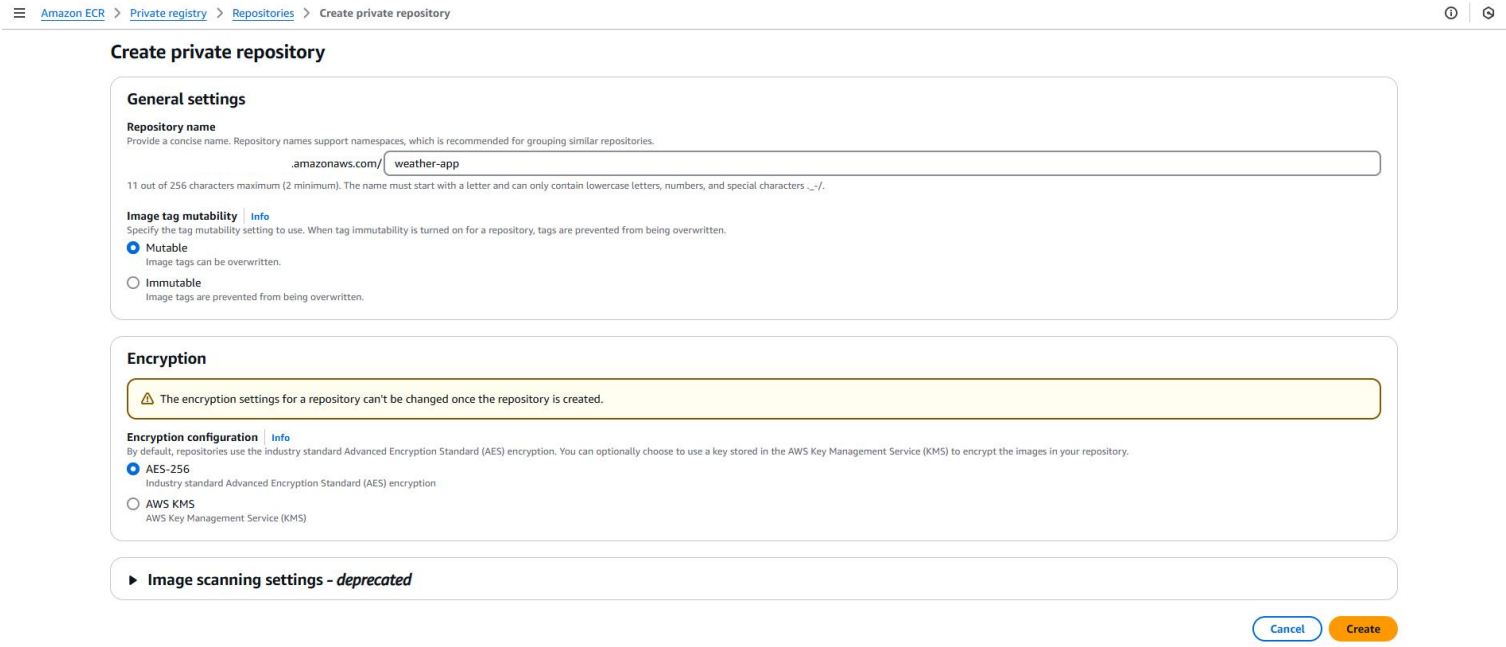
In the Deploy Stage, AWS CodeDeploy updates the Amazon Elastic Container Service (ECS) with the new Docker image from ECR. ECS runs the containers, ensuring the Node.js app is operational, scalable, and ready to handle requests. Traffic is routed through the Application Load Balancer (ALB), which distributes incoming requests evenly across the ECS tasks, providing high availability and reliability for end users.

Step-by-Step implementation

NB: - All the necessary files are on the repository given above.
- The creation of the app files isn't covered here as it isn't our topic.
- Make sure that you have the latest version of the AWS CLI and Docker installed.
- AWS CLI must be configured.

Step1: Create an ECR repository to store docker images:

Amazon ECR is a fully managed Docker container registry that stores the Docker image securely. It allows versioning of images, ensuring that the correct image is used during deployment.
We go to Amazon ECR service and we click on **create repository**, then we fill in the necessary informations as it is shown below



We click on 'create'. Note that we are creating a private registry.
We should then go to the CLI and use the following steps to authenticate and push an image to our repository:

1. Retrieve an authentication token and authenticate Docker client to our registry. Use the AWS CLI command:
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com

2. Build a Docker image using the following command:
docker build -t weather-app .

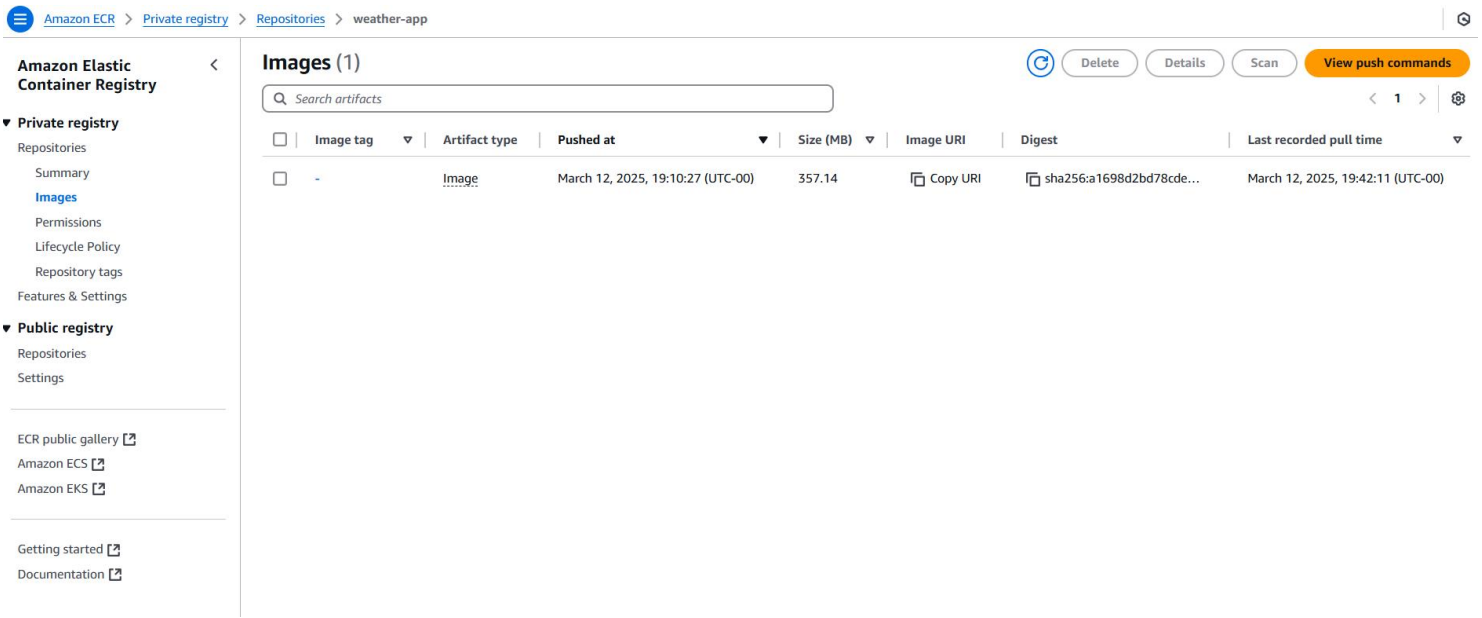
3. After the build completes, we tag our image so we can push the image to this repository:
docker tag weather-app:latest {YOUR_ACCOUNT_ID}.dkr.ecr.{YOUR_REGION}.amazonaws.com/weather-app:latest

4. Run the following command to push this image to the newly created AWS repository:
docker push {YOUR_ACCOUNT_ID}.dkr.ecr.{YOUR_REGION}.amazonaws.com/weather-app:latest

After these 4 steps we should see that the image is been pushing on the repository like this:

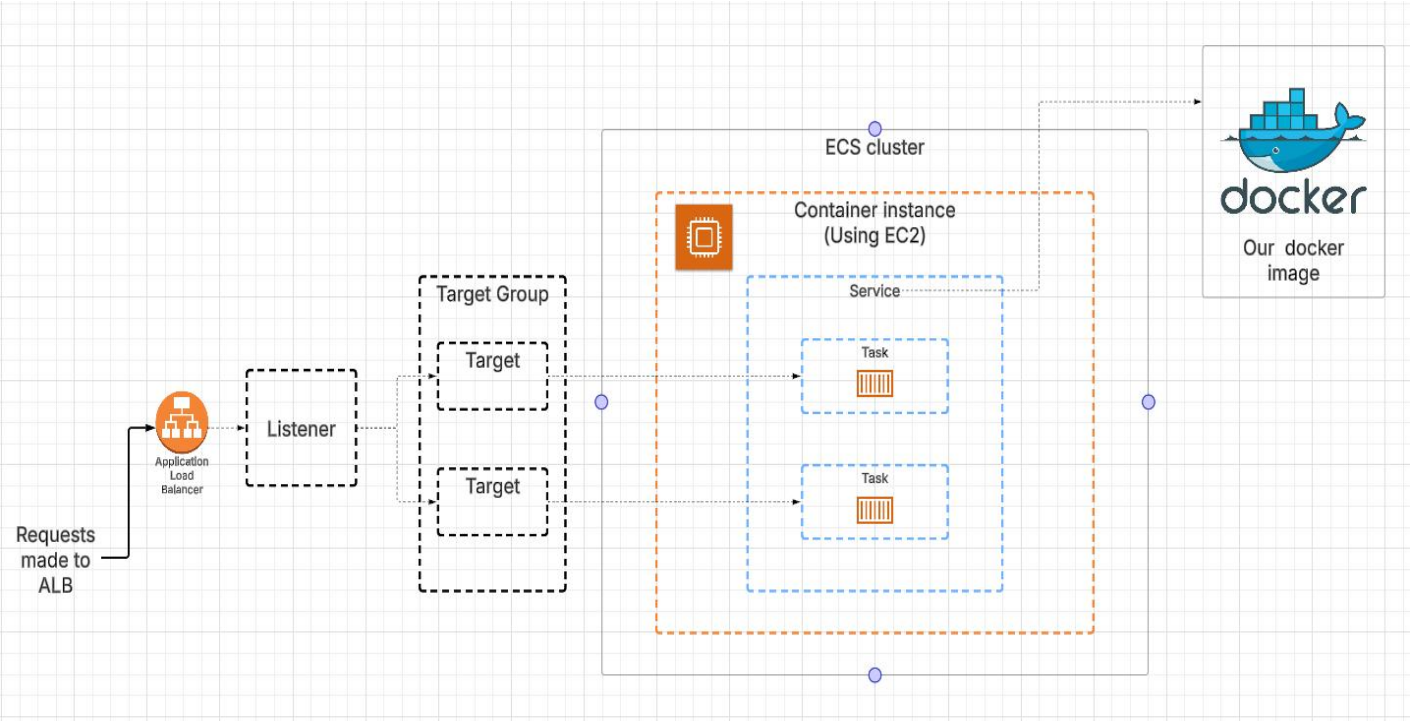
```
0738bb988510: Pushed
bc4e4e530d16: Pushed
c2771b4263da: Pushed
0319727accbd: Pushed
0d5f5a015e5d: Pushed
3c777d951de2: Pushed
f8a91dd5fc84: Pushing [============================>                ]  62.35MB/100.4MB
cb81227abde5: Pushed
e01a454893a9: Pushing [=>                                           ]  15.92MB/509.5MB
c45660adde37: Pushing [=============>                               ]  41.02MB/145.6MB
fe0fb3ab4a0f: Pushed
f1186e5061f2: Pushed
b2dba7477754: Pushing [=====>                                       ]  11.98MB/114.1MB
```

We should wait till the upload is complete, then, on the console we should see the image uploaded to the repository:



Step 2: Create an ECS Task, ECS Cluster, ECS service and configure the load balancer:

The Application Load Balancer (ALB) and Amazon ECS work together to ensure our containerized application, which uses a Docker image, is scalable, highly available, and accessible to users. The ALB acts as the entry point for incoming traffic, using listeners to accept requests on specific ports (e.g., HTTP on port 80 or HTTPS on port 443) and forwarding them to a target group. The target group is associated with our ECS service, which manages the tasks (containers) running our Dockerized application. ECS automatically registers the tasks with the target group, allowing the ALB to distribute traffic evenly across them. The ALB also performs health checks to ensure only healthy tasks receive traffic. If a task fails, ECS replaces it, maintaining the desired number of running tasks. This seamless integration ensures our application is always available, scalable, and responsive to user requests.
The diagram below illustrates the functioning of these services.



First, we create 2 security groups one for the application load balancer and one for securing the traffic from the application load balancer to ECS, we go to the EC2 service, choose security groups, click on **create security group** and we fill in the necessary informations:

Create security group Info

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a new security group, complete the fields below.

Basic details

Security group name Info

weather-app-alb-sg

Name cannot be edited after creation.

Description Info

this is the security group for our application load balancer

VPC Info

vpc-0bb0d666ceb0a46d6

Inbound rules Info

Type Info	Protocol Info	Port range Info	Source Info		Description - optional Info	
HTTP	TCP	80	Anywhe...	0.0.0.0/0		Delete

Add rule

Outbound rules Info

Type Info	Protocol Info	Port range Info	Destination Info		Description - optional Info	
All traffic	All	All	Anywhe...	0.0.0.0/0		Delete

Add rule

Tags - *optional*

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

No tags associated with the resource.

Add new tag

You can add up to 50 more tags

Cancel Create security group

Create security group Info

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a new security group, complete the fields below.

Basic details

Security group name Info

weather-app-alb-to-ecs-sg

Name cannot be edited after creation.

Description Info

this is the security group allowing traffic from elb to ecs

VPC Info

vpc-0bb0d666ceb0a46d6

Inbound rules Info

Type Info	Protocol Info	Port range Info	Source Info		Description - optional Info	
All TCP	TCP	0 - 65535	Custom	sg-02f2695568662b473 sg-02f2695568662b473		Delete

Add rule

Outbound rules Info

Type Info	Protocol Info	Port range Info	Destination Info		Description - optional Info	
All traffic	All	All	Custom	0.0.0.0/0		Delete

Add rule

⚠ Rules with destination of 0.0.0.0/0 or ::/0 allow your instances to send traffic to any IPv4 or IPv6 address. We recommend setting security group rules to be more restrictive and to only allow traffic to specific known IP addresses.

Tags - *optional*

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

No tags associated with the resource.

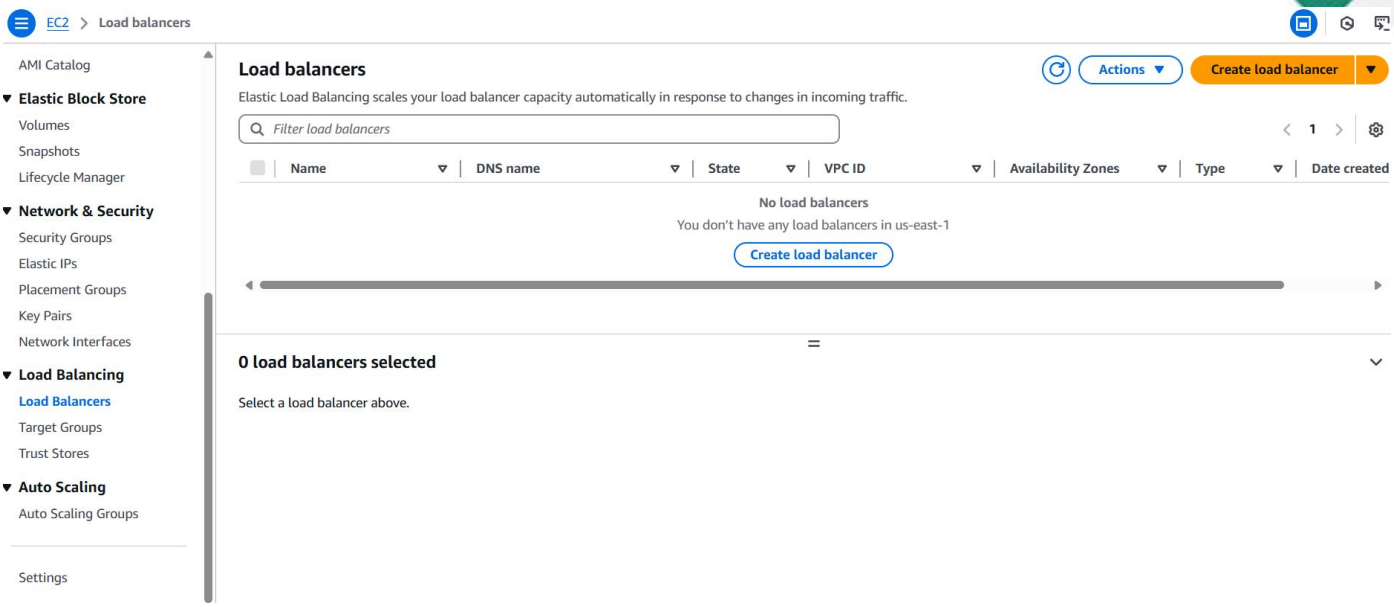
Add new tag

You can add up to 50 more tags

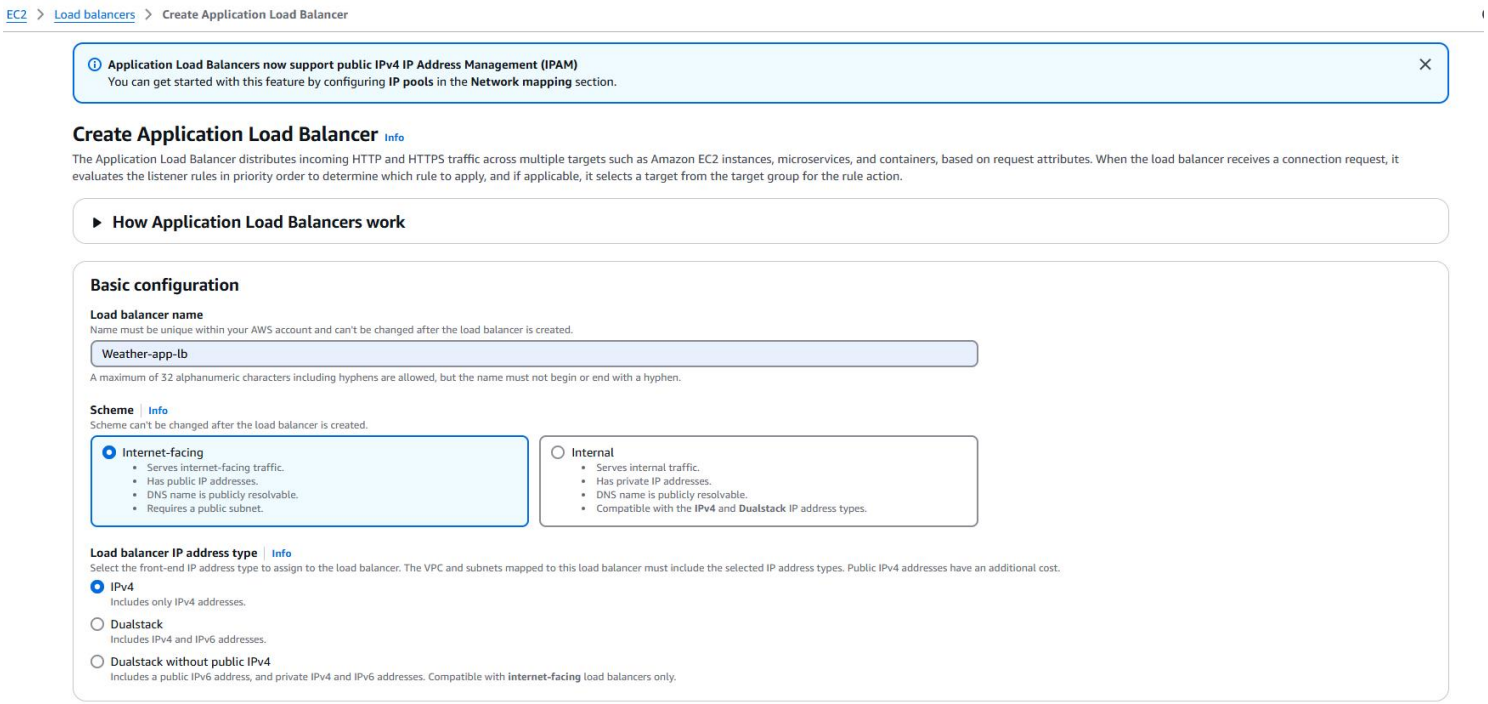
Cancel Create security group

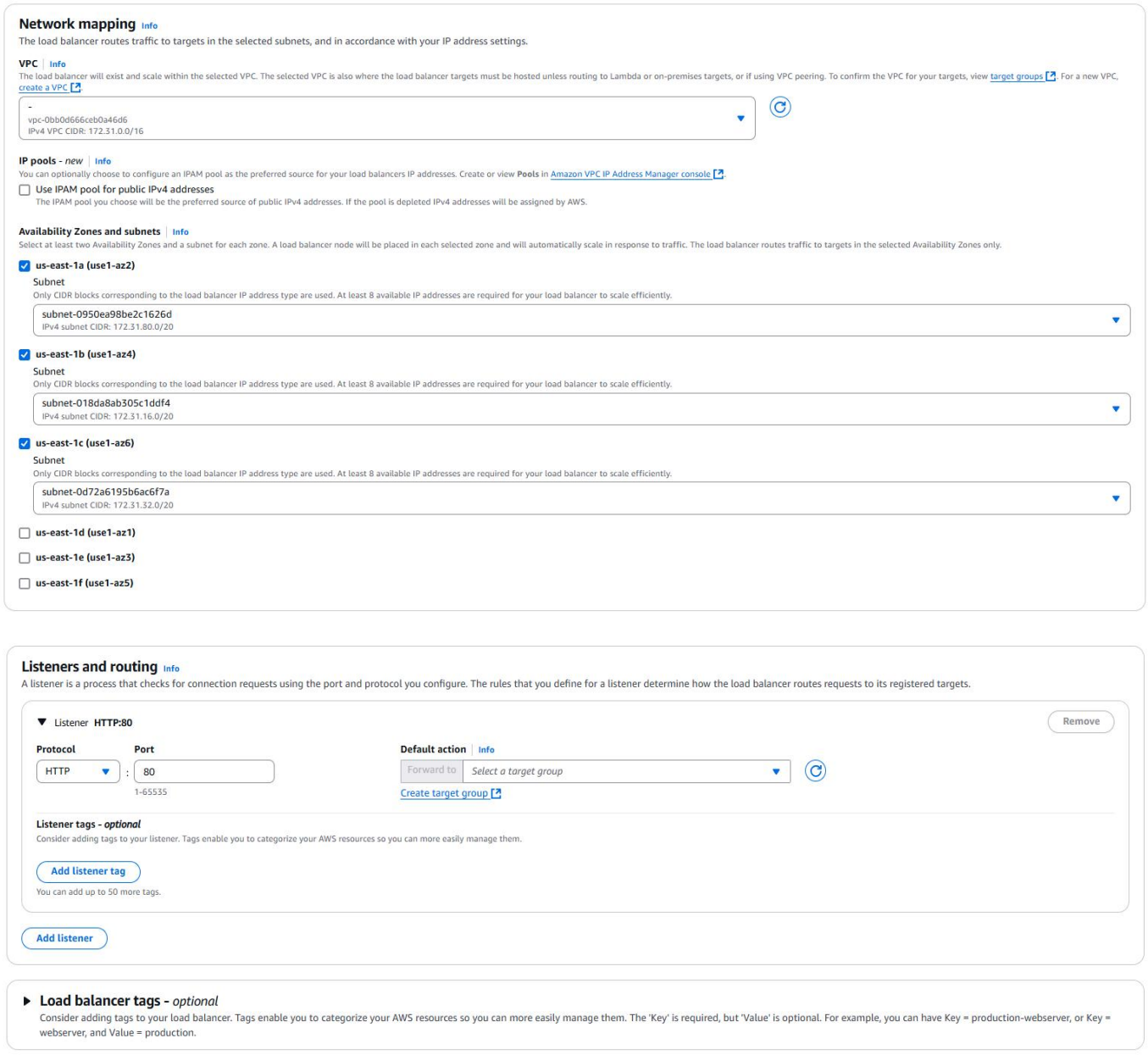
Note that the **weather-app-alb-sg** is on the source of this security group

We'll create now an ALB, we go to EC2 on the loadBalancers tab, click on create load Balancer

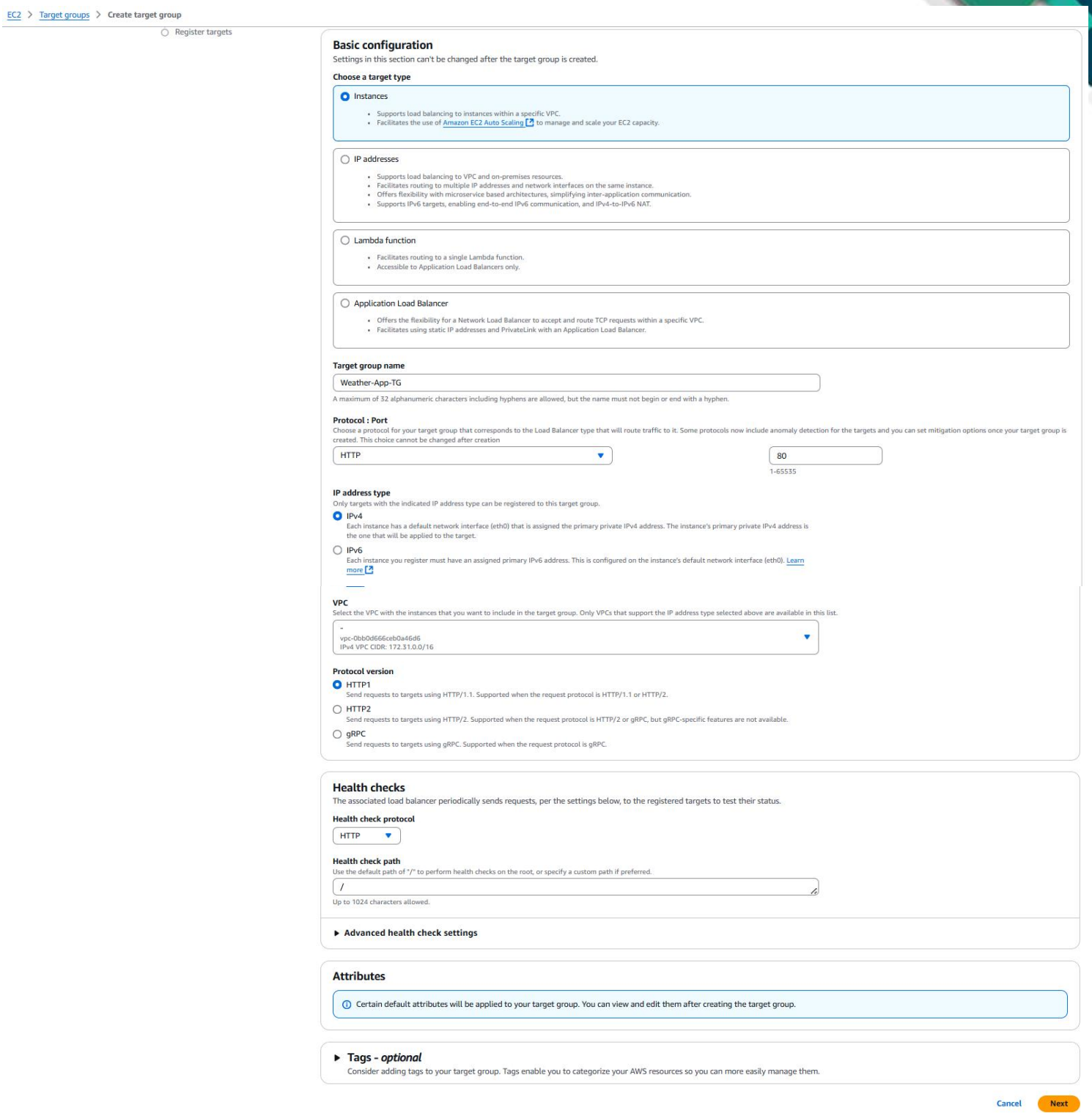


We select **Application Load Balancer** and we fill in the necessary informations:

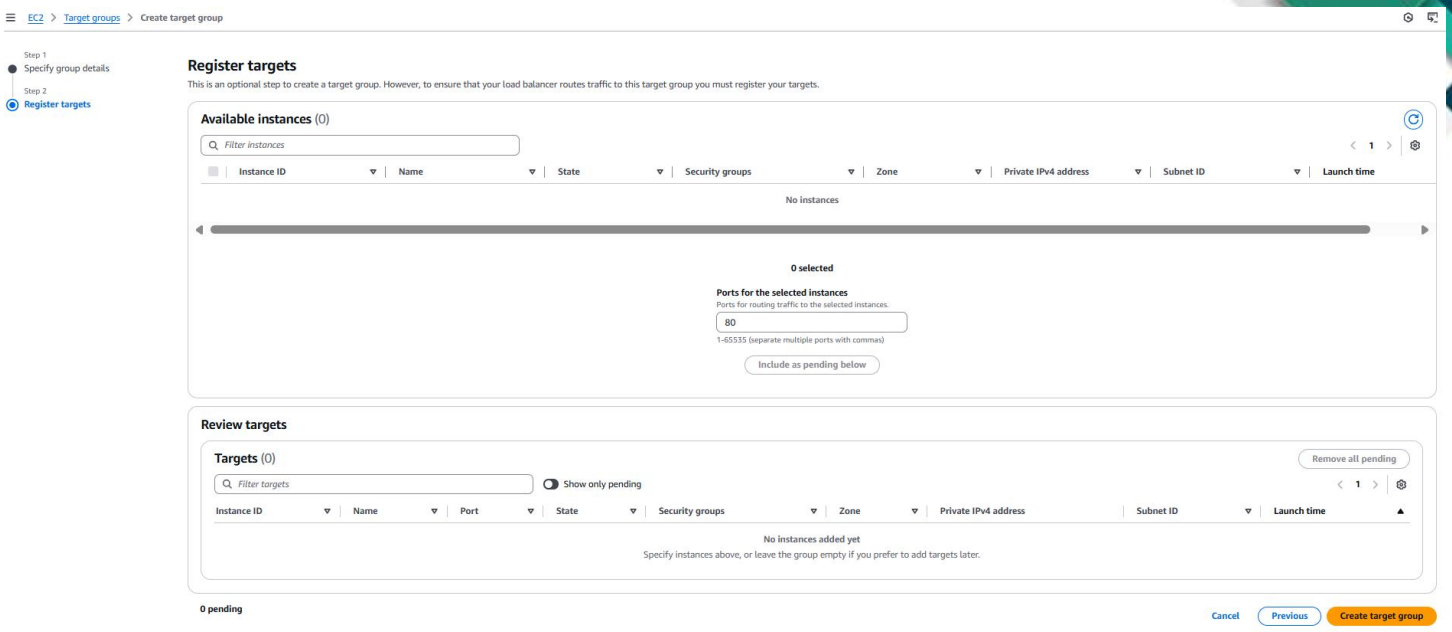




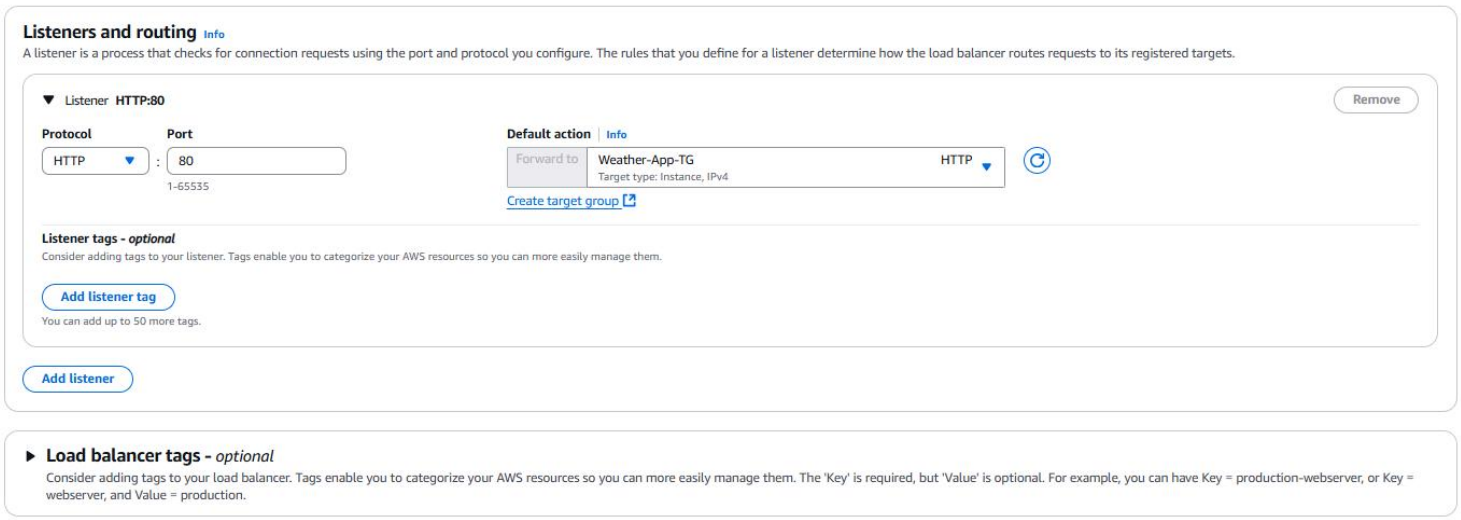
We click on **Create target group**
Another page will appear to create the target group and we fill in the necessary informations:

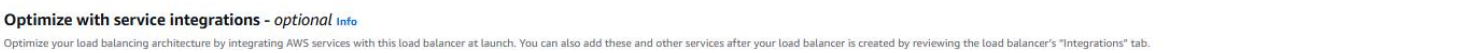


We click on ‘Next’

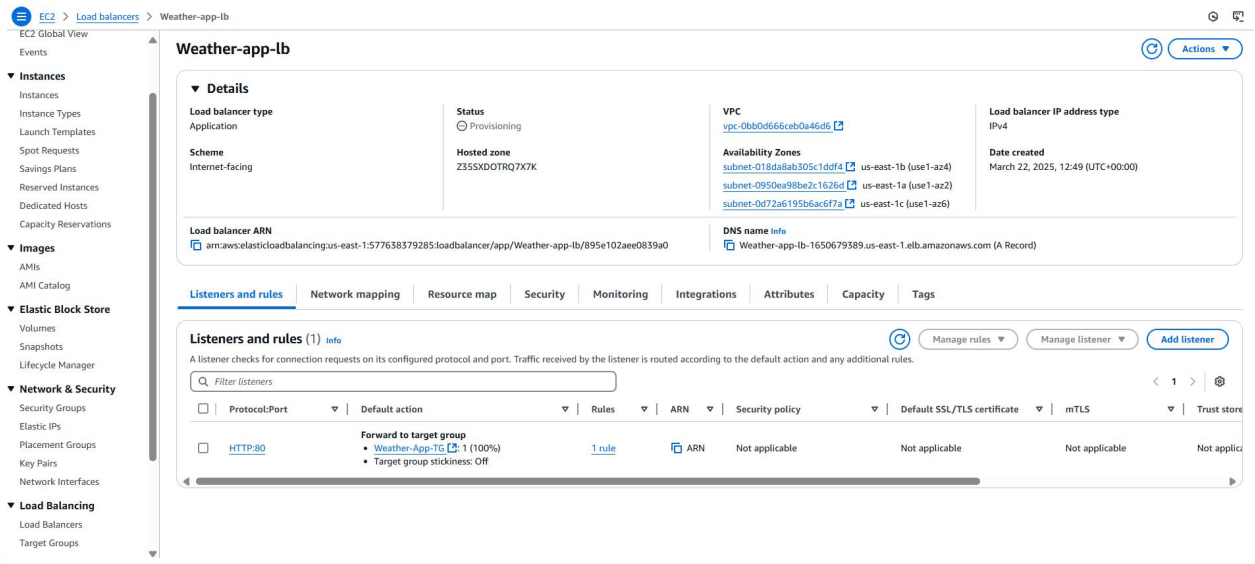


We click on Create ‘target group’
Back to our ALB creation, we refresh an select our target group created earlier



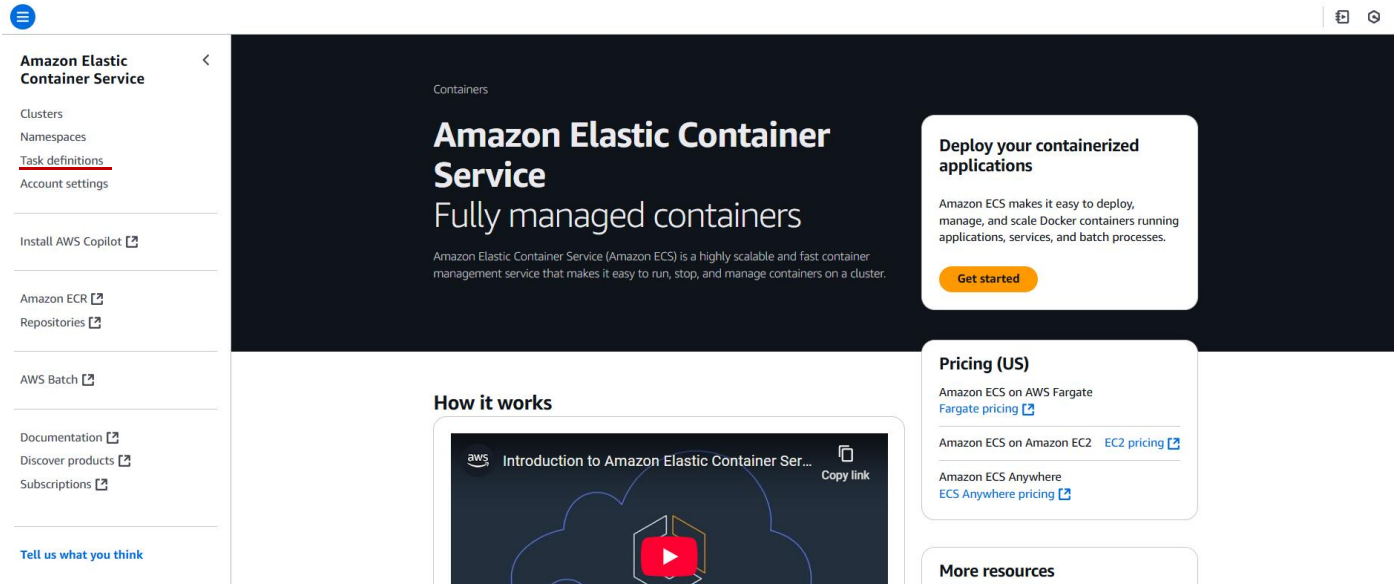


We leave all the other things as they are and we click on ‘create load balancer’,we should get this:

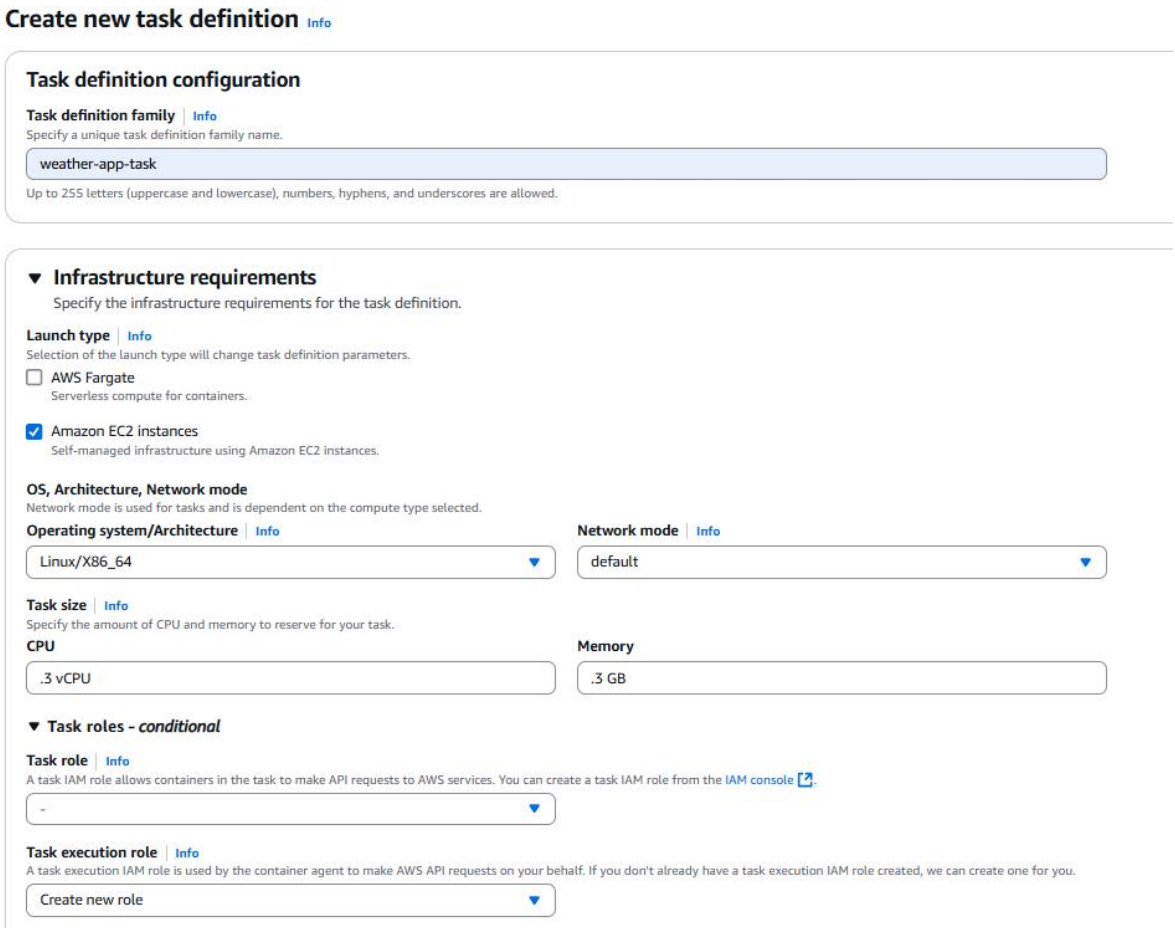


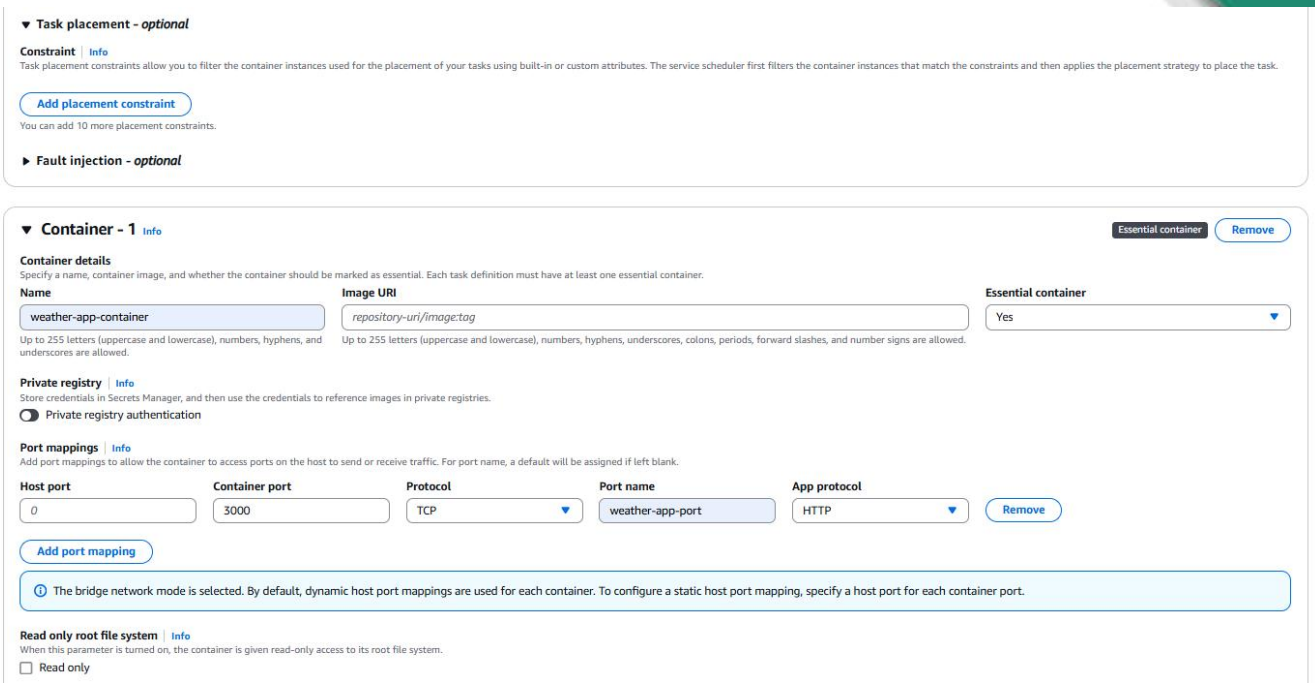
Now, we're going to create an ECS task, which is a blueprint for our application. It includes details such as the Docker image to use (from ECR), CPU and memory requirements, networking settings, and environment variables. This step ensures that ECS knows how to run our Node.js application container.

We go to ECS, we click on the task definitions tab



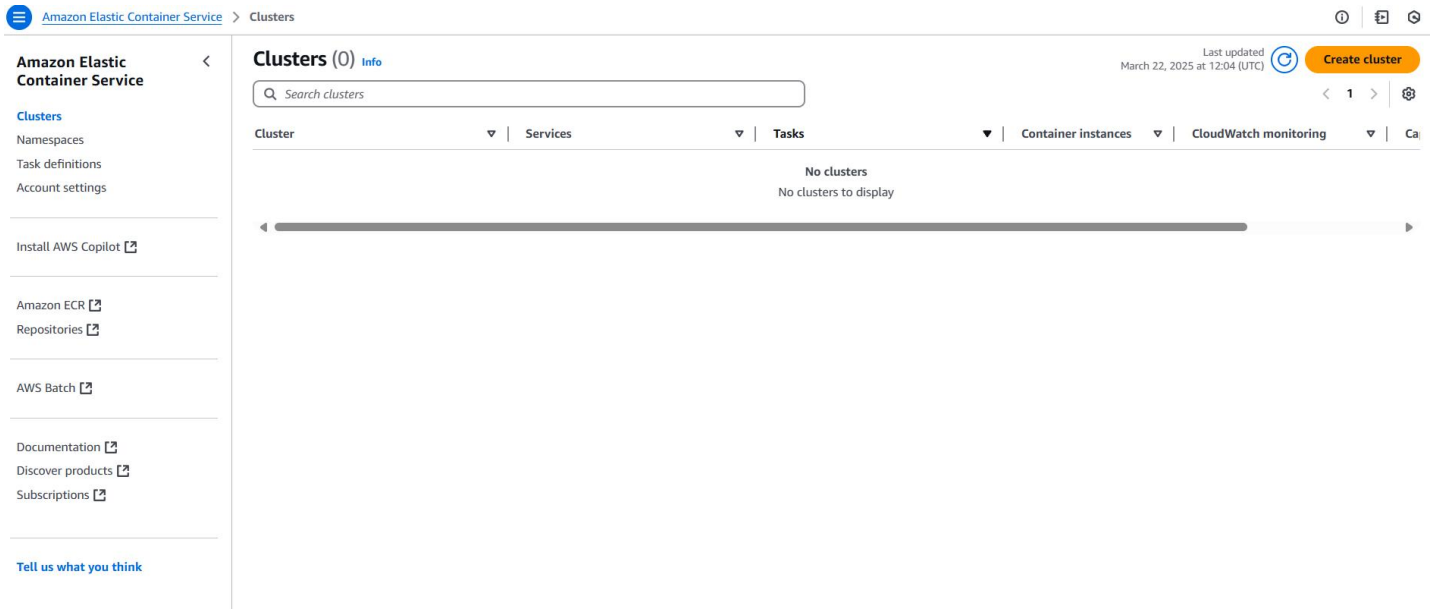
We click on **create a new task definition**, we fill in the informations like it is shown below:



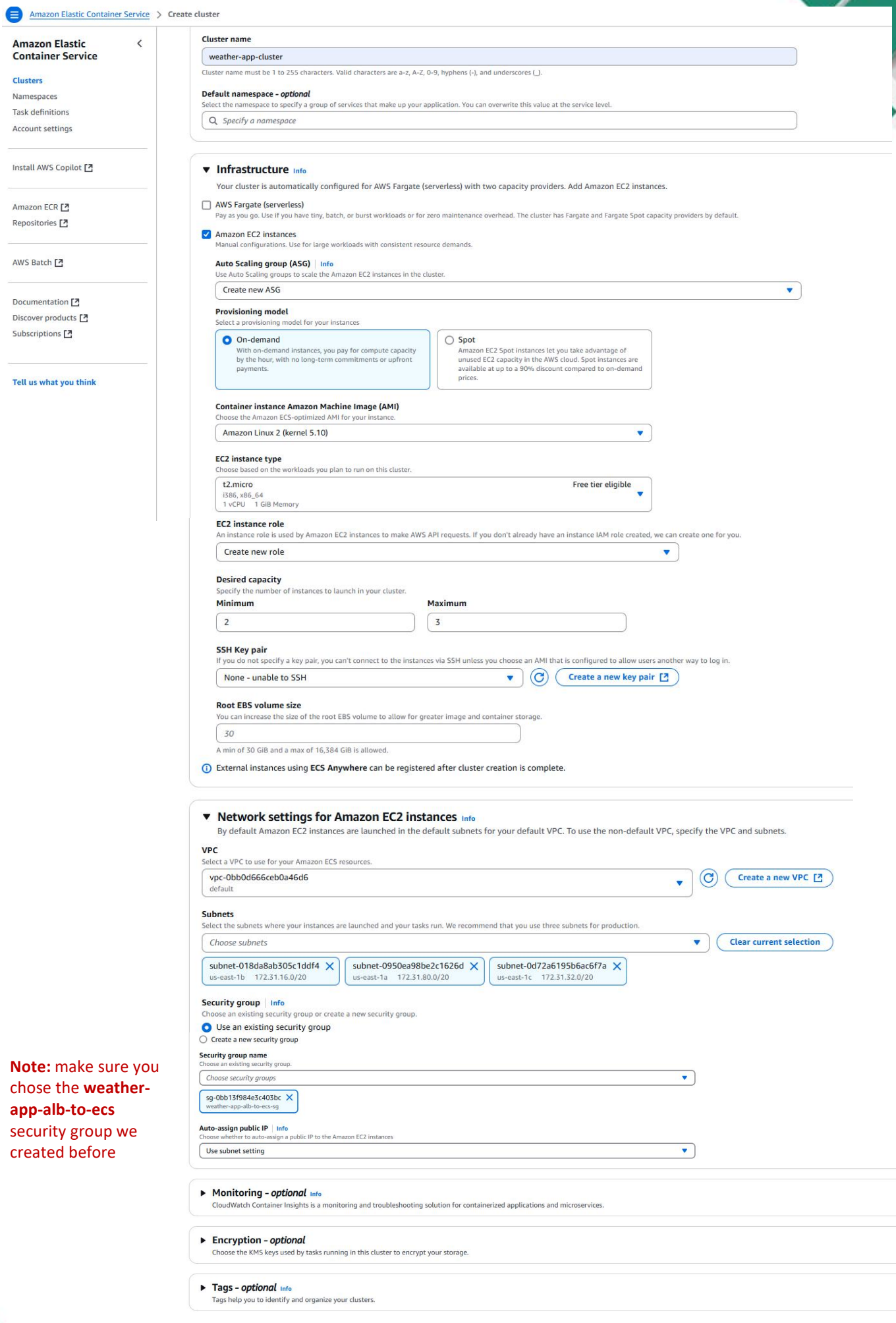


Note: In the Image URI field, copy and paste the one created in the ECR repository during step1 and add ":latest" to it.
We live all the other configurations as they are and we click on **create.**

Next up, we're gonna create an ECS cluster which is a logical grouping of resources (EC2 instances or Fargate tasks) where your containers will run. The cluster provides the infrastructure needed to deploy and manage our Node.js application. We can choose to use EC2 instances for more control or AWS Fargate for a serverless approach, here we'll choose EC2.



We go to ECS, click on 'Create cluster'

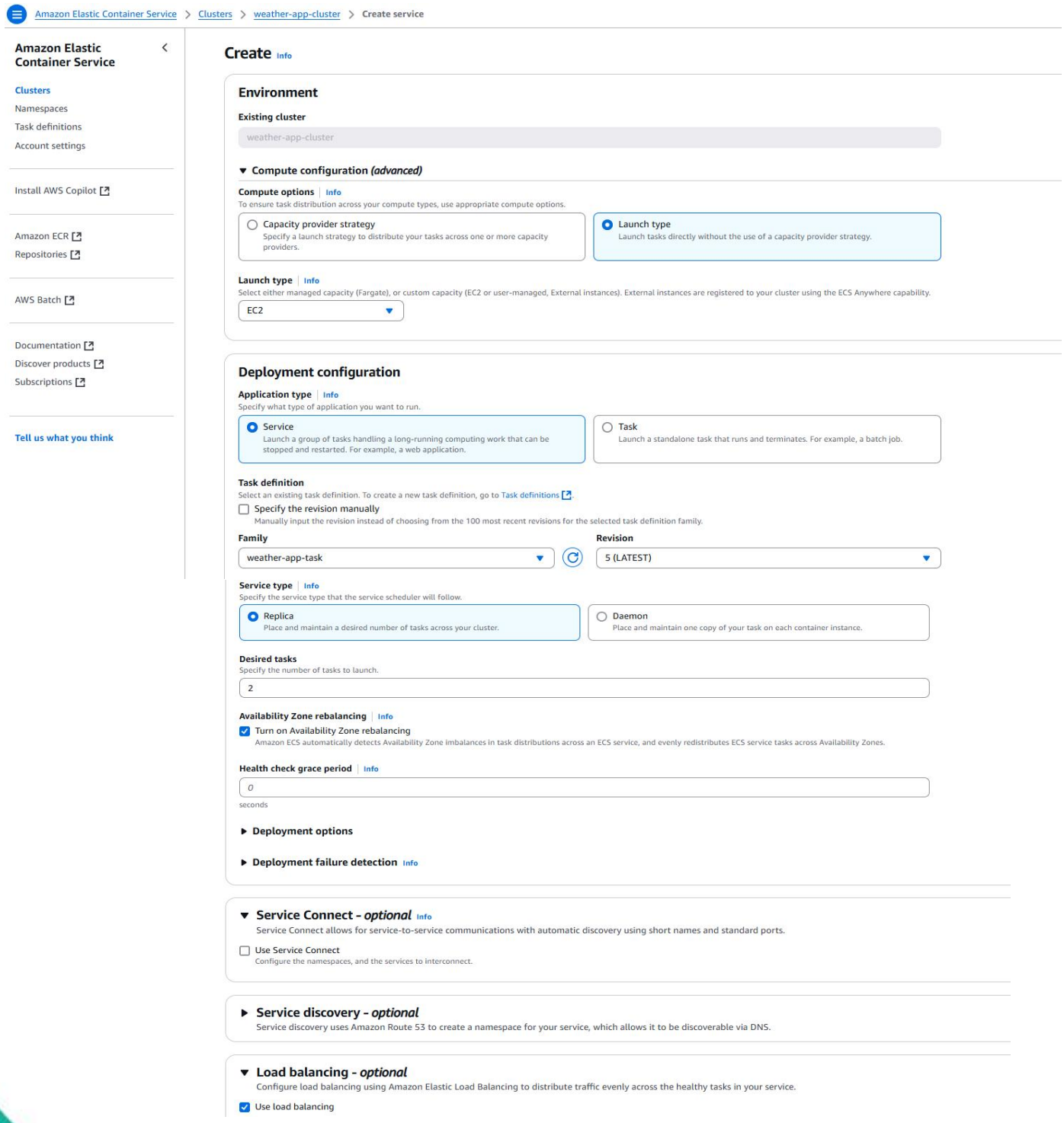


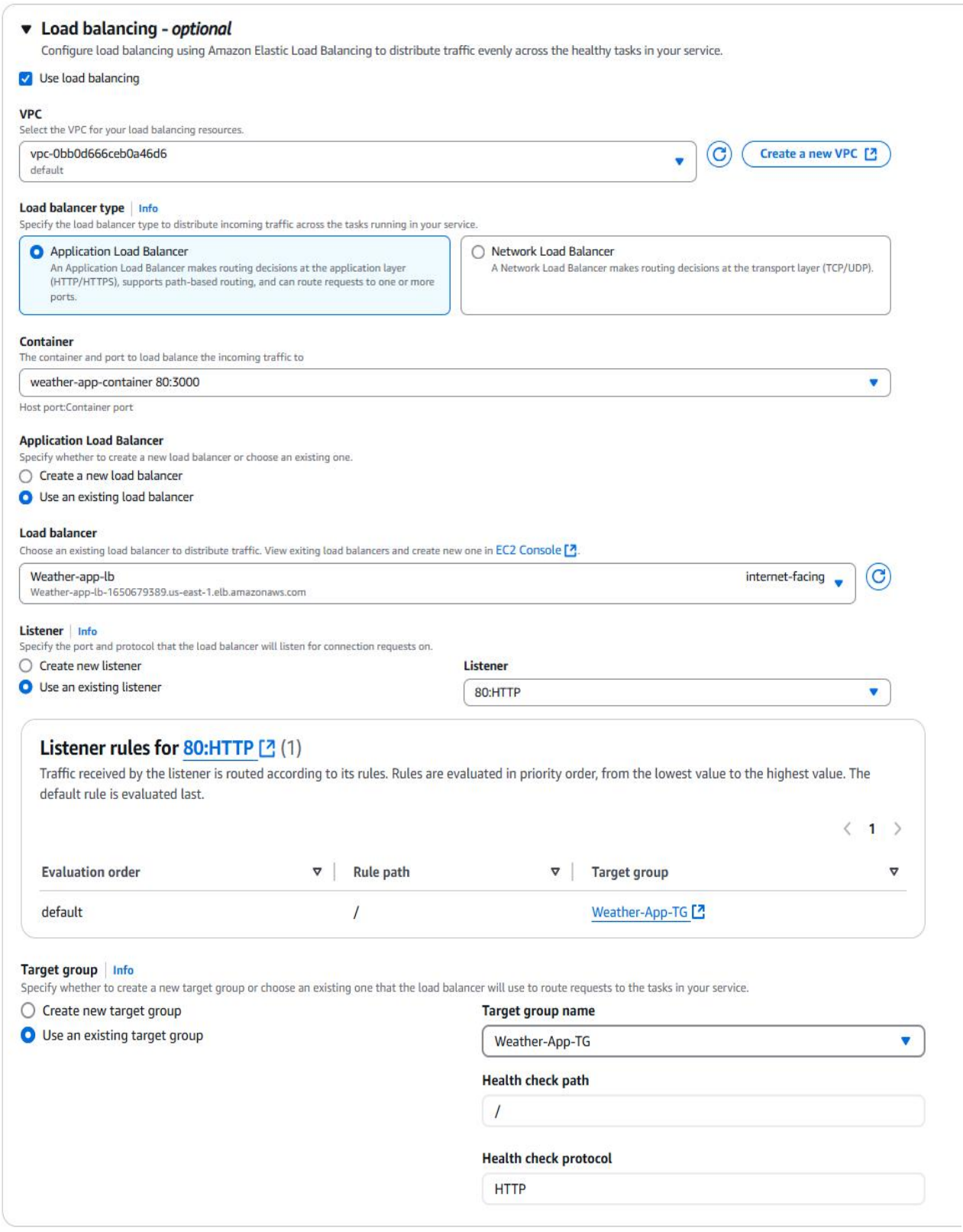
Note: make sure you chose the **weather-app-alb-to-ecs** security group we created before

After filling all the informations, click on ‘create’

We’ll now walk through the process of creating an ECS Service, a core feature of an Amazon ECS cluster that helps us manage and maintain our containerized applications. The main goal of an ECS Service is to ensure that a specific number of tasks (containers) are always running and healthy. It provides us with scalability, high availability, and the ability to seamlessly update our application, making it easier to deploy and manage workloads in a production environment. Let’s dive into the steps to create an ECS Service.

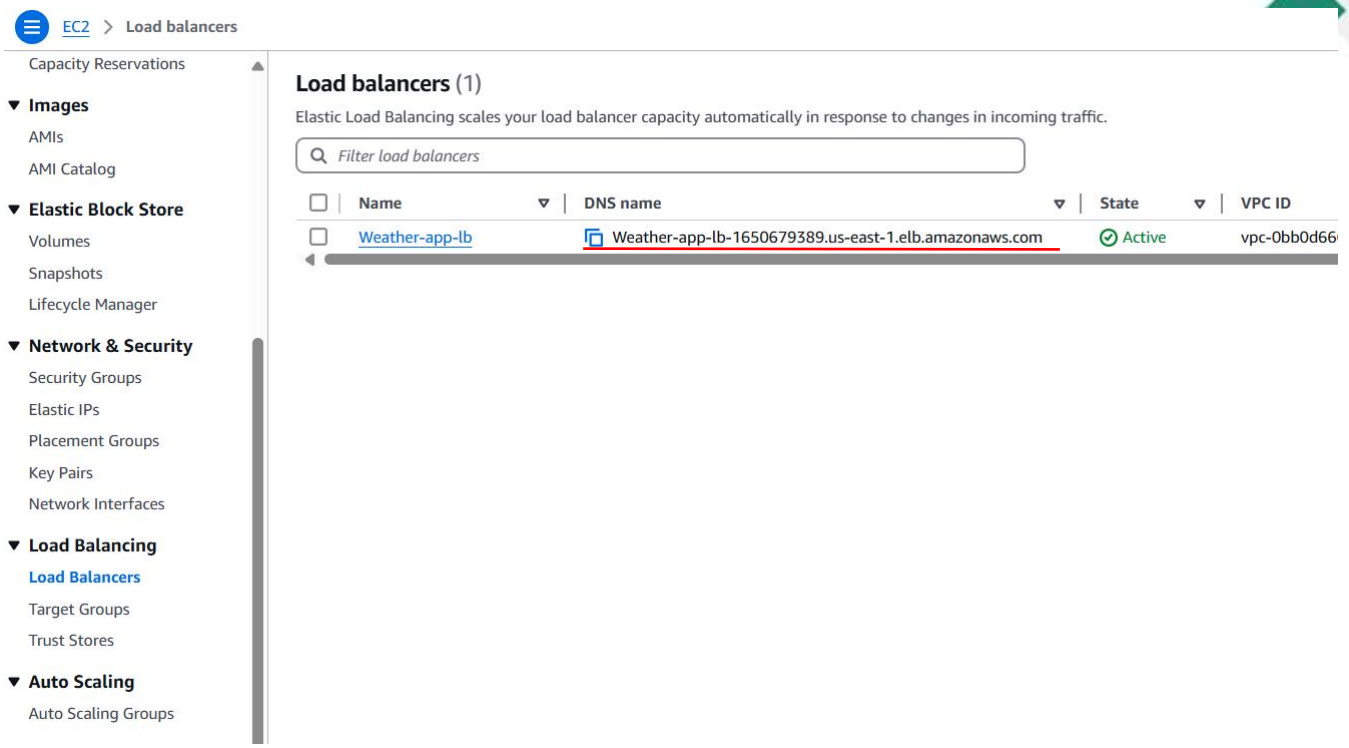
So we click on the **weather-app-cluster** created before, go to services tab, click on ‘create’ and fill in the necessary informations:



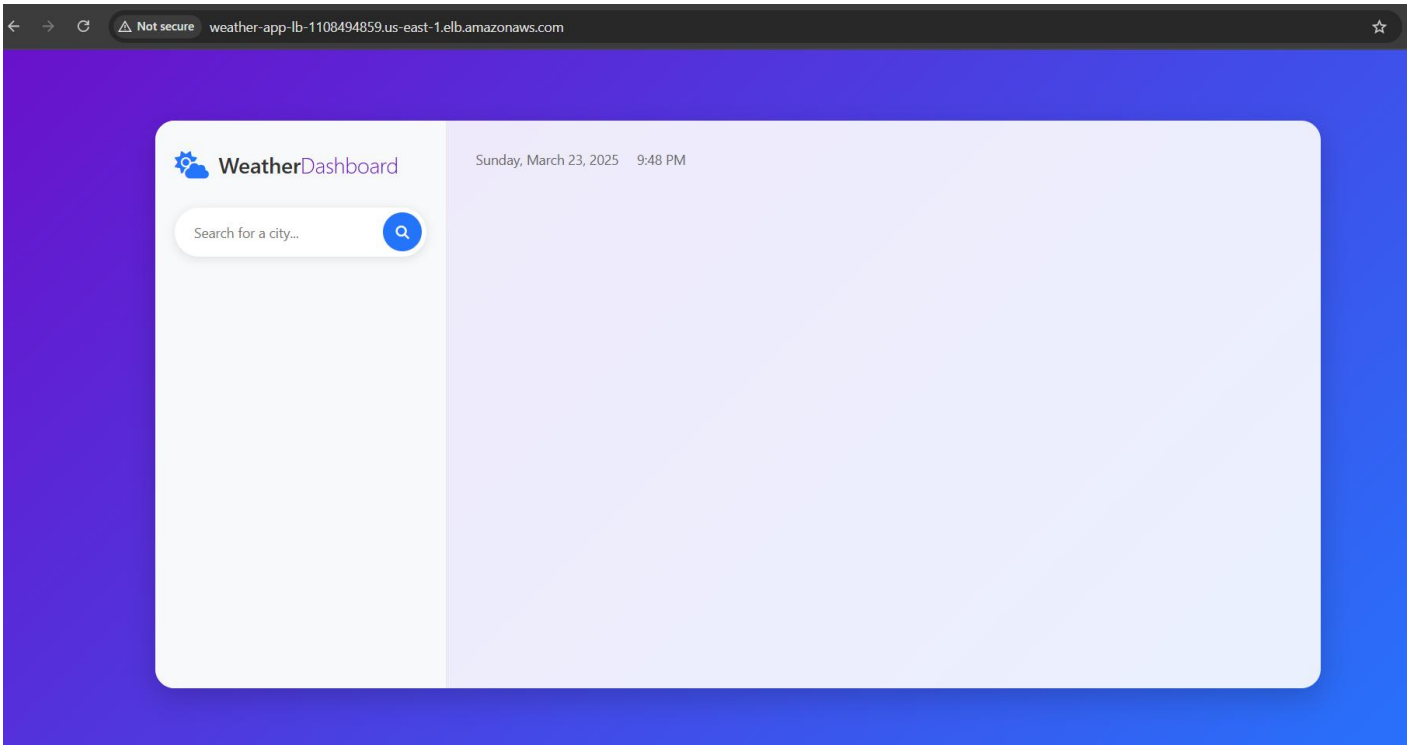


We live all the other configurations as they are and we click on **create**

Once it has launched after some waiting time, it's time to put it to the test. We go to the load balancer, copy it DNS name:

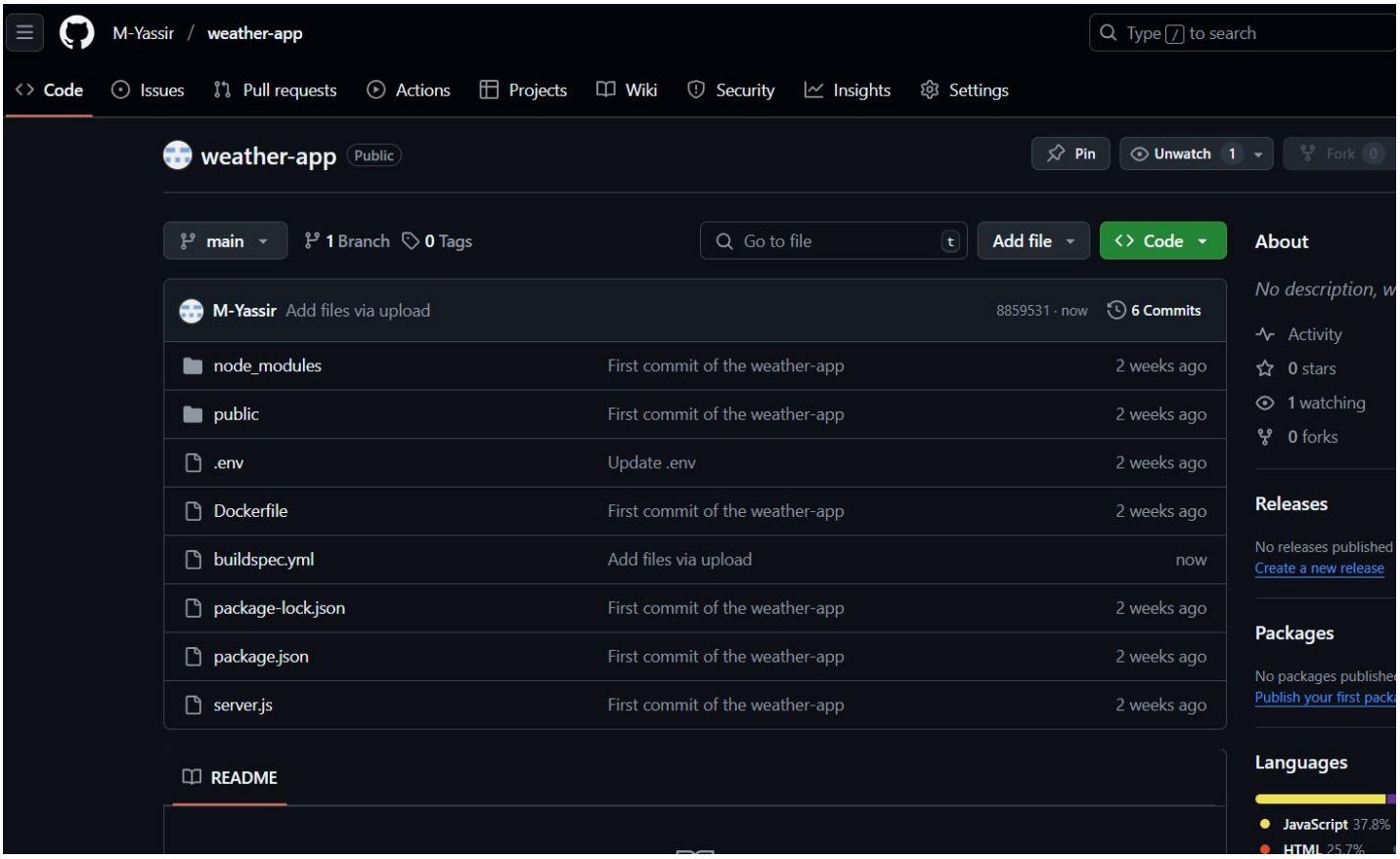


Paste it on the browser and we should get this:



Step3: Configure the CICD pipeline:

First, we'll upload the application files to GitHub and include a **buildspec.yml** file which defines the build process for our application in AWS CodeBuild, specifying steps like installing dependencies, building the Docker image, and pushing it to Amazon ECR. By adding it to our repository, CodeBuild automatically executes these steps when changes are pushed, streamlining our CI/CD pipeline and ensuring consistent builds, when the files are uploaded, the repo should look like this:



Then, we'll create a build project in code build, we go to codebuild, click on 'create project' and fill the necessary informations

Note: make sure you connect your github account to aws, you can do it when creating the build project by clicking on Manage account credentials

Developer Tools > CodeBuild > Build projects > Create build project

Create build project

Project configuration

Project name

weather-app-project

A project name must be 2 to 255 characters. It can include the letters A-Z and a-z, the numbers 0-9, and the special characters - and _.

Project type

Select what type of project you would like to create. Info

- (•) Default project
  Create a custom CodeBuild project.
- ( ) Runner project
  Create a CodeBuild managed runner for workflows in GitHub Actions, GitHub Enterprise Actions, GitLab, or Buildkite.

▶ **Additional configuration**
Description, public build access, build badge, concurrent build limit, tags

▼ Source

Add source

Source 1 - Primary

Source provider

GitHub

Credential

Your account is successfully connected by using an AWS managed GitHub App. Manage account credentials.

[ ] Use override credentials for this project only

Repository

- (•) Repository in my GitHub account
- ( ) Public repository
- ( ) GitHub scoped webhook

Repository

https://github.com/M-Yassir/weather-app

Source version - *optional* Info

Enter a pull request, branch, commit ID, tag, or reference and a commit ID.

▶ **Additional configuration**
Git clone depth, Git submodules, Build status config

Primary source webhook events Info

Webhook - *optional* Info

[ ] Rebuild every time a code change is pushed to this repository

▼ Environment

Provisioning model Info

- (•) On-demand
  Automatically provision build infrastructure in response to new builds.
- ( ) Reserved capacity
  Use a dedicated fleet of instances for builds. A fleet's compute and environment type will be used for the project.

Environment image

- (•) Managed image
  Use an image managed by AWS CodeBuild
- ( ) Custom image
  Specify a Docker image

Compute

- (•) EC2
  Optimized for flexibility during action runs
- ( ) Lambda
  Optimized for speed and minimizes the start up time of workflow actions

Running mode

- (●) Container — Running on Docker container
- ( ) Instance — Running on EC2 instance directly

Operating system

Ubuntu

Runtime(s)

Standard

Image

aws/codebuild/standard:7.0

Image version

Always use the latest image for this runtime version

[ ] Use GPU-enhanced compute

Service role

- (●) New service role — Create a service role in your account
- ( ) Existing service role — Choose an existing service role from your account

Role name

code-build-weather-app-role

Type your service role name

▼ **Additional configuration**

Timeout, privileged, certificate, VPC, compute type, environment variables, file systems, auto-retry, registry credential

Auto-retry limit

CodeBuild will automatically call retry build using the project's service role up to the auto-retry limit

0

Timeout

Default timeout is 1 hour

Hours: 1 | Minutes: 0

Timeout must be between 5 minutes and 36 hours

Queued timeout

Default time in build queue is 8 hours

Hours: 8 | Minutes: 0

Timeout must be between 5 minutes and 8 hours

Privileged

[x] Enable this flag if you want to build Docker images or want your builds to get elevated privileges

Report auto-discover Info

[ ] Disable report auto-discover

Auto-discover directory - *optional*

**/*

CodeBuild will search for supported report file types in this directory. **/* by default

Certificate

If you have a self-signed certificate or a certificate signed by a certification authority, choose the option to install it from your S3 bucket.

- (●) Do not install any certificate
- ( ) Install certificate from your S3 bucket

VPC

Select a VPC that your AWS CodeBuild project will access.

Compute

- (●) 3 GB memory, 2 vCPUs
- ( ) 7 GB memory, 4 vCPUs
- ( ) 15 GB memory, 8 vCPUs

○ 70 GB memory, 36 vCPUs
○ 145 GB memory, 72 vCPUs

Registry credential - *optional*

You can supply a secret name or ARN from AWS Secrets Manager to authenticate to an external registry. The credentials can be used for image overrides from self-hosted runners.

Environment variables

Name	Value	Type	
		Plaintext	Remove

Add environment variable

Create parameter

File systems

Identifier	ID	
		Remove

Directory path - *optional* | Mount point

Mount options - *optional*

Add file system

Buildspec

Build specifications

Build specifications

○ Insert build commands
Store build commands as build project configuration

● Use a buildspec file
Store build commands in a YAML-formatted buildspec file

Buildspec name - *optional*

By default, CodeBuild looks for a file named buildspec.yml in the source code root directory. If your buildspec file uses a different name or location, enter its path from the source root here (for example, buildspec-two.yml or configuration/buildspec.yml).

buildspec.yml

Batch configuration

You can run a group of builds as a single execution. Batch configuration is also available in advanced option when starting build.

☐ Define batch configuration - *optional*
You can also define or override batch configuration when starting a build batch.

Artifacts

Add artifact

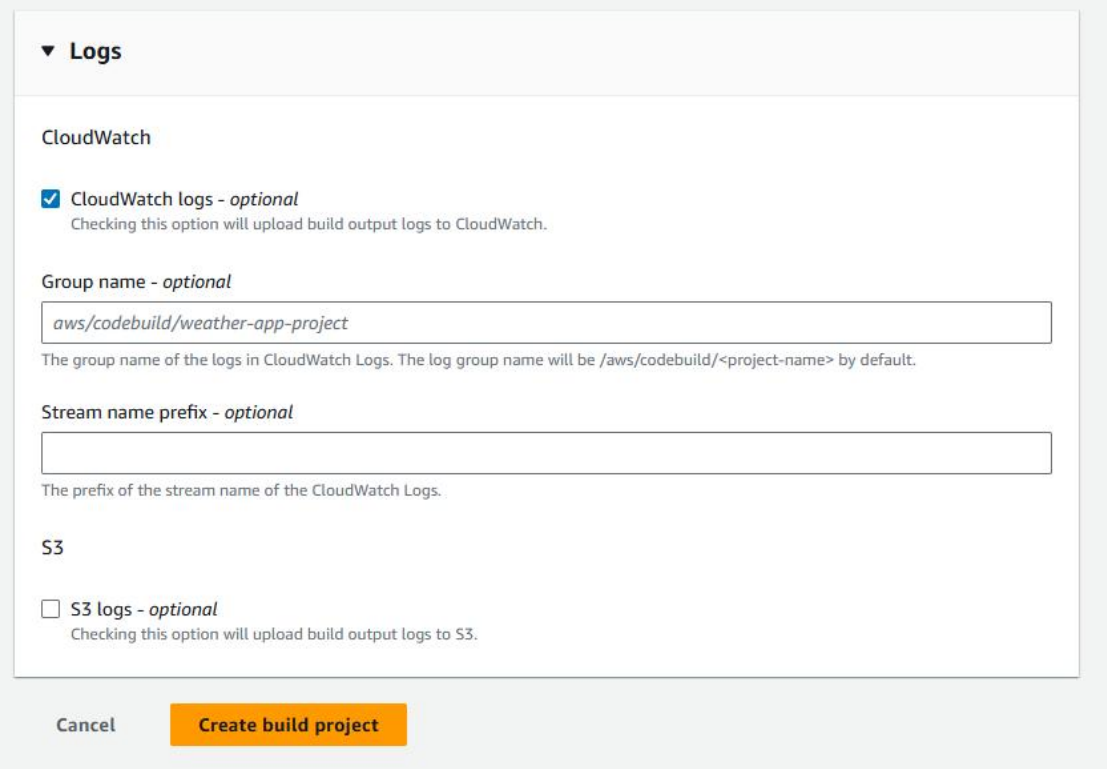
Artifact 1 - Primary

Type

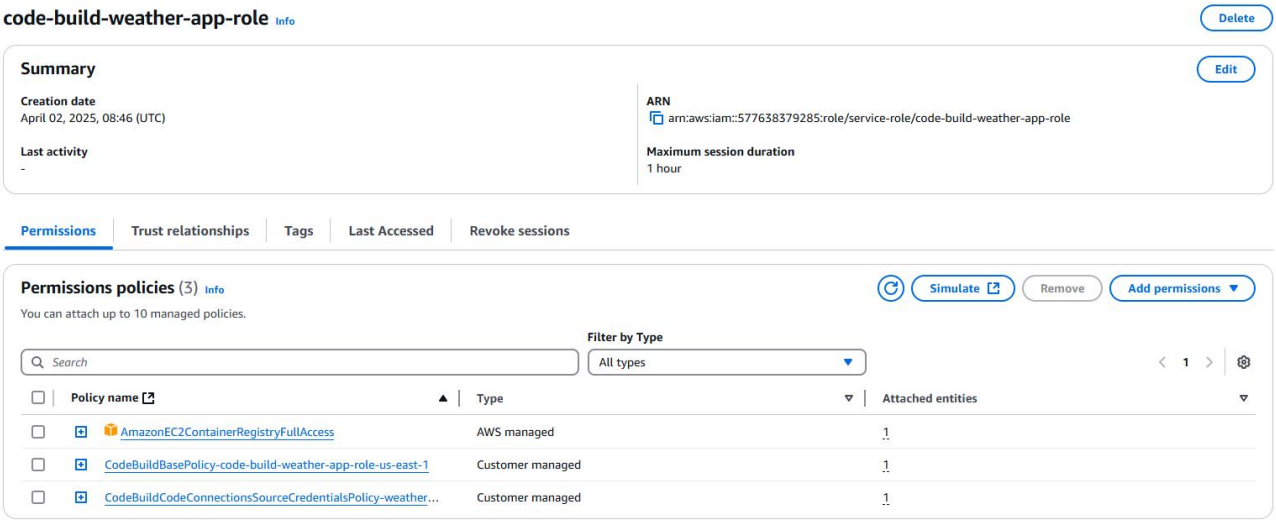
No artifacts

You might choose no artifacts if you are running tests or pushing a Docker image to Amazon ECR.

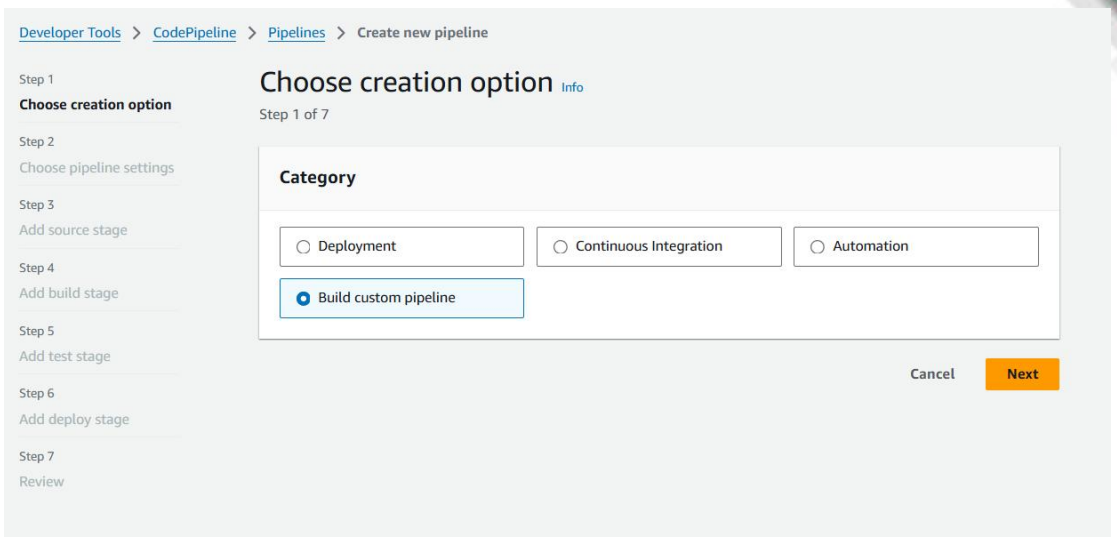
▶ **Additional configuration**
Cache, encryption key

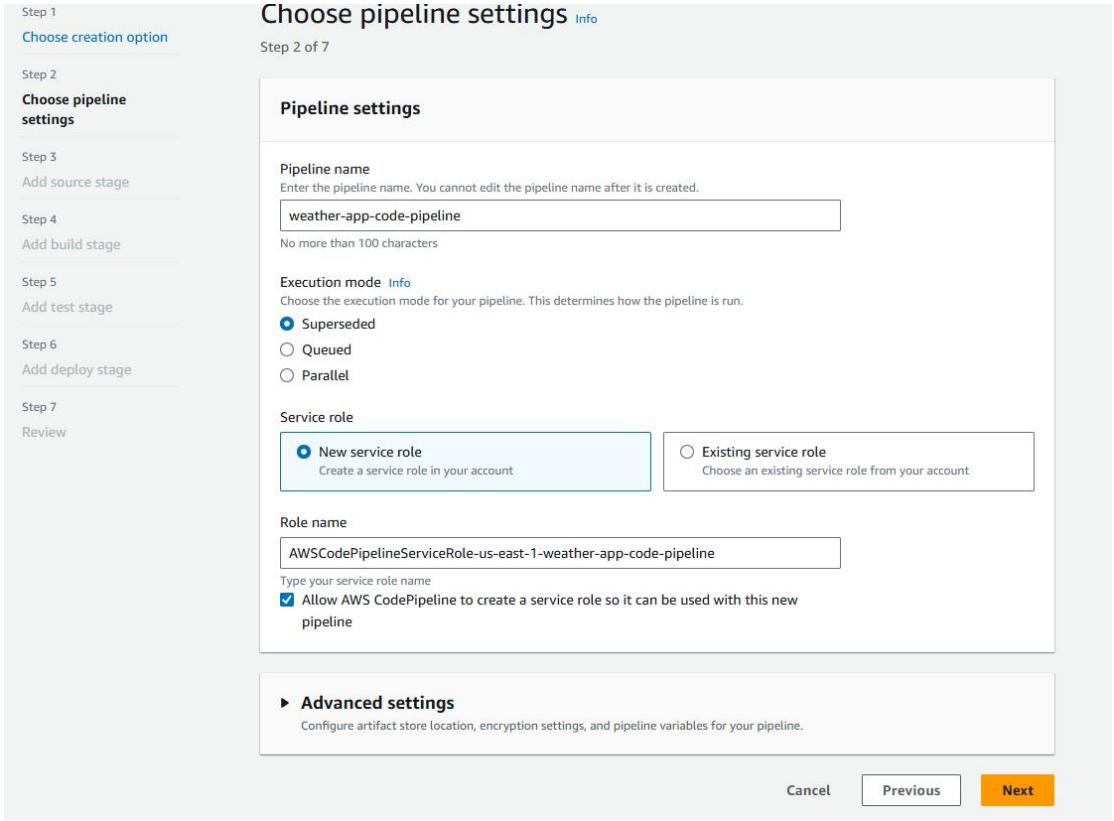


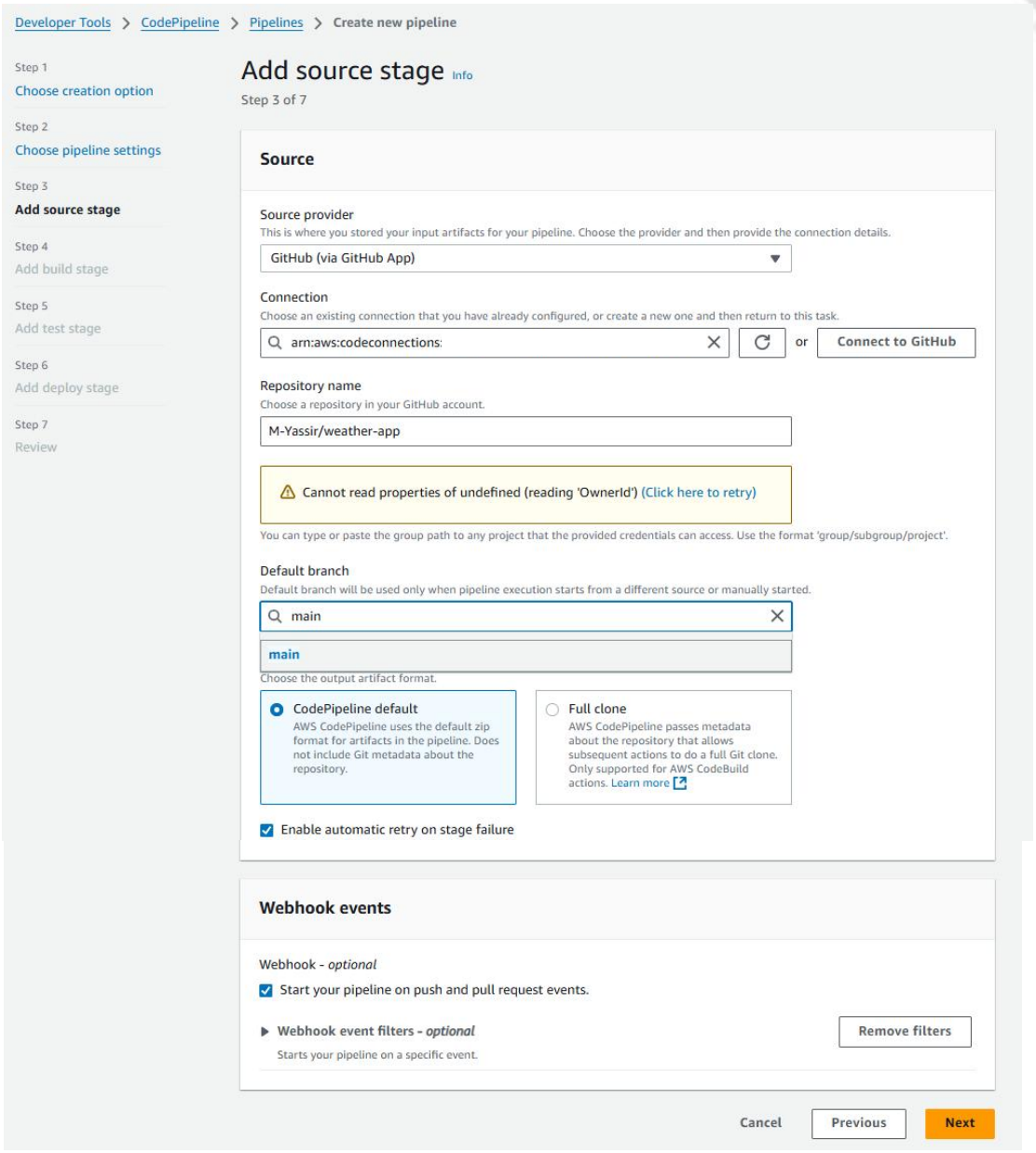
After creating the build project, you must add an **AmazonEC2ContainerRegistryFullAccess** permission to the policy given to the code-build-weather-app-role like that:

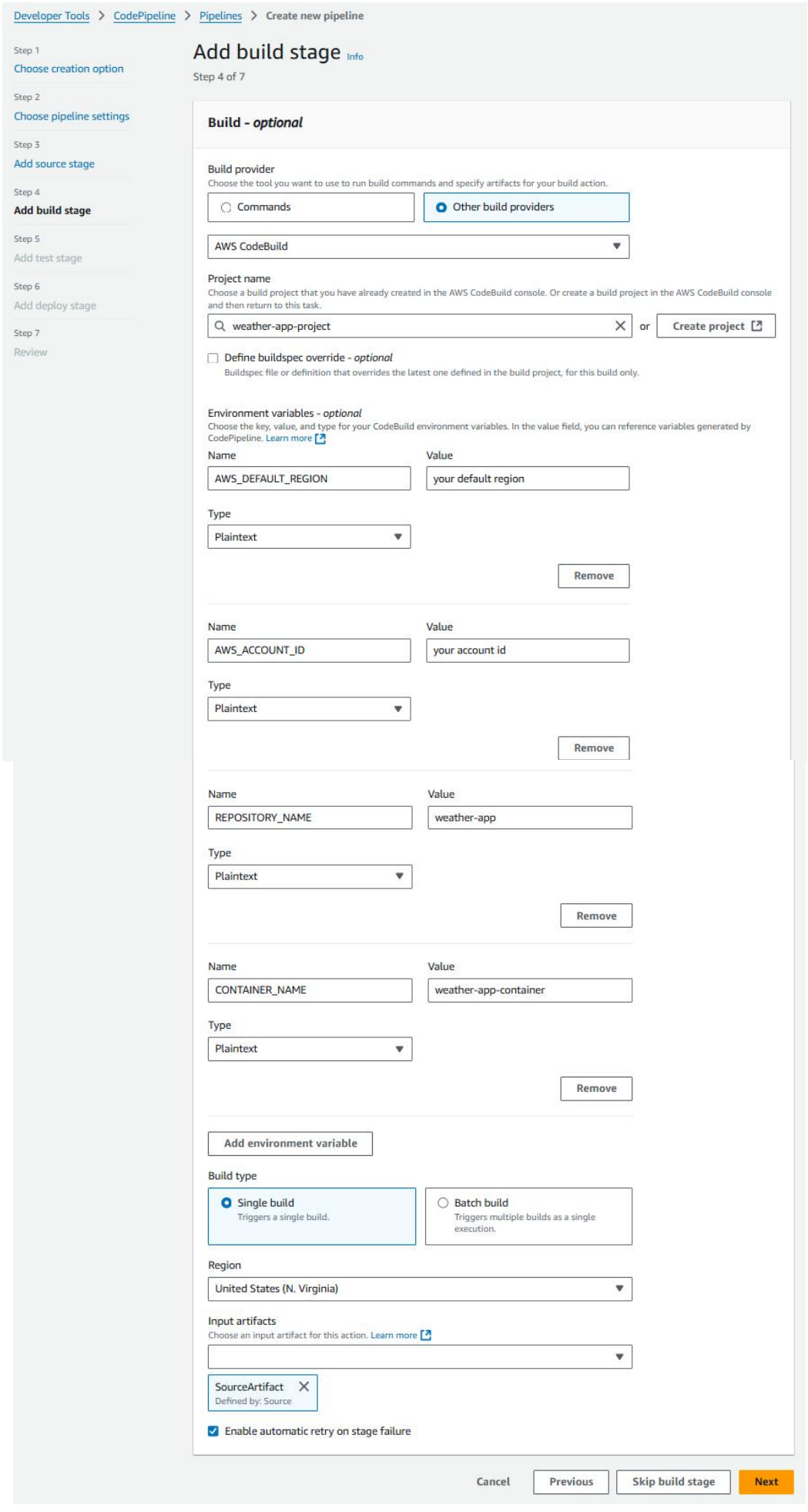


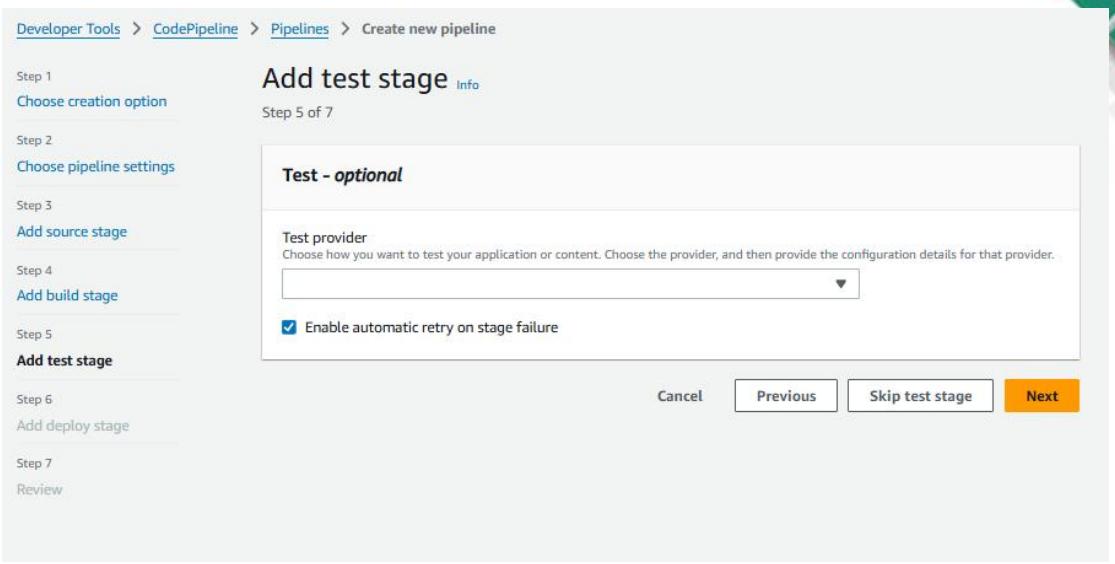
Finally, configuring the pipeline, we go to codepipeline, click on **create pipeline** and fill in the necessary informations:

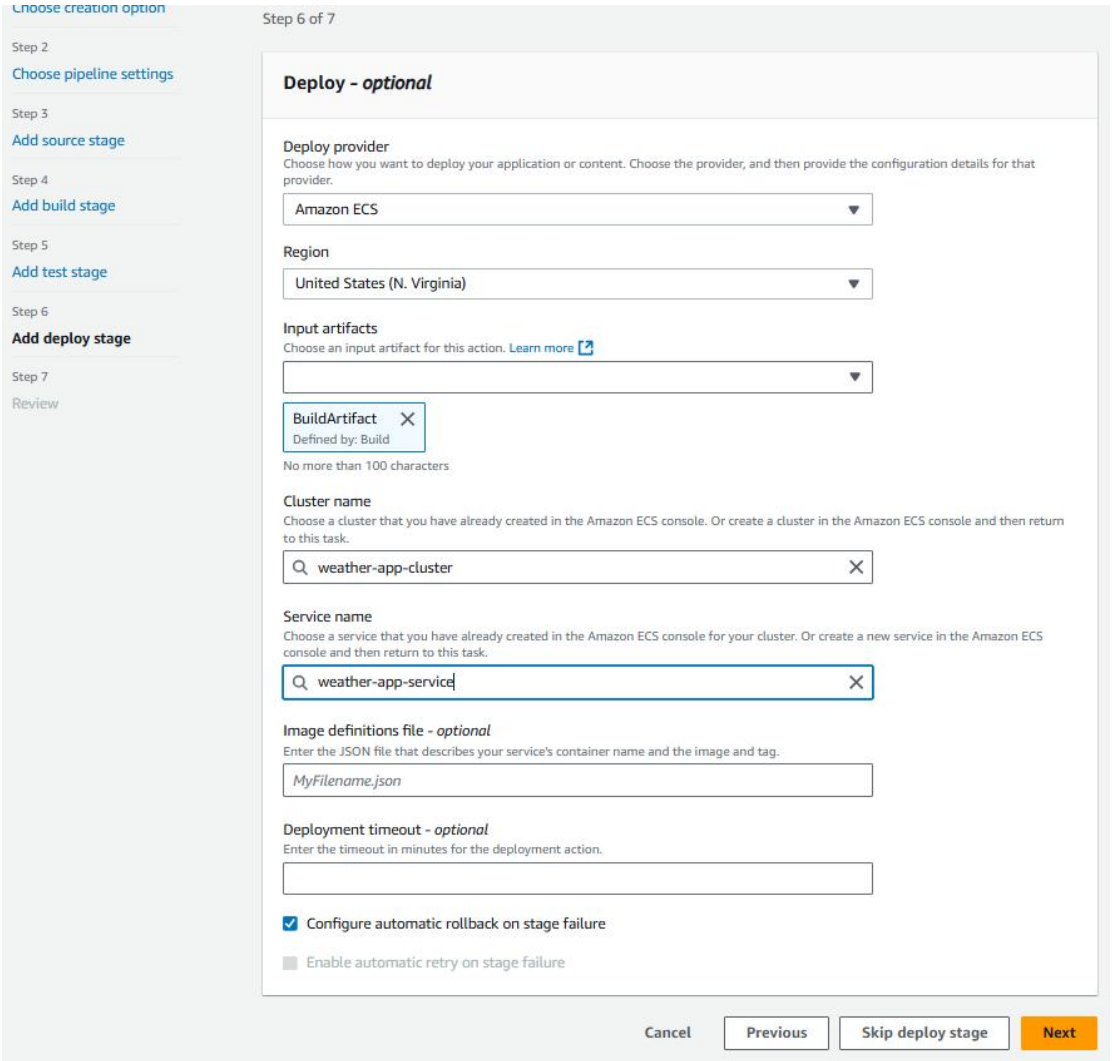

Developer Tools > CodePipeline > Pipelines > Create new pipeline
Step 1
Choose creation option
Step 2
Choose pipeline settings
Step 3
Add source stage
Step 4
Add build stage
Step 5
Add test stage
Step 6
Add deploy stage
Step 7
Review
Choose creation option Info
Step 1 of 7
Category
Deployment
Continuous Integration
Automation
Build custom pipeline
Cancel
Next


Step 1
Choose creation option
Step 2
Choose pipeline settings
Step 3
Add source stage
Step 4
Add build stage
Step 5
Add test stage
Step 6
Add deploy stage
Step 7
Review
Choose pipeline settings Info
Step 2 of 7
Pipeline settings
Pipeline name
Enter the pipeline name. You cannot edit the pipeline name after it is created.
weather-app-code-pipeline
No more than 100 characters
Execution mode Info
Choose the execution mode for your pipeline. This determines how the pipeline is run.
Superseded
Queued
Parallel
Service role
New service role
Create a service role in your account
Existing service role
Choose an existing service role from your account
Role name
AWSCodePipelineServiceRole-us-east-1-weather-app-code-pipeline
Type your service role name
Allow AWS CodePipeline to create a service role so it can be used with this new pipeline
Advanced settings
Configure artifact store location, encryption settings, and pipeline variables for your pipeline.
Cancel
Previous
Next

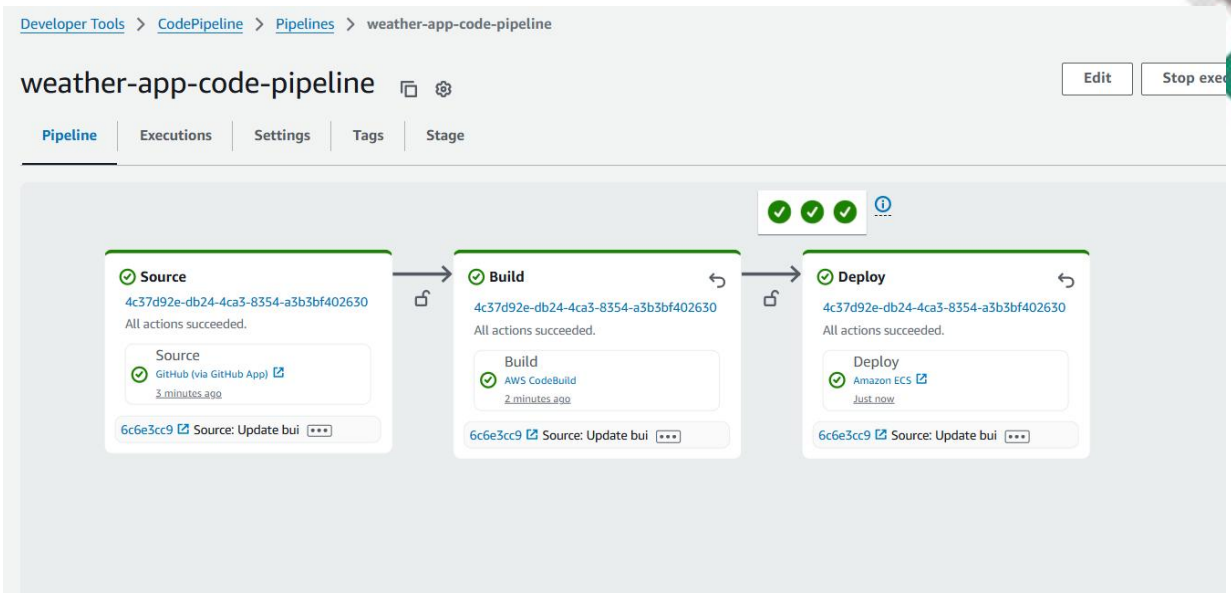

Developer Tools > CodePipeline > Pipelines > Create new pipeline
Step 1
Choose creation option
Step 2
Choose pipeline settings
Step 3
Add source stage
Step 4
Add build stage
Step 5
Add test stage
Step 6
Add deploy stage
Step 7
Review
Add source stage Info
Step 3 of 7
Source
Source provider
This is where you stored your input artifacts for your pipeline. Choose the provider and then provide the connection details.
GitHub (via GitHub App)
Connection
Choose an existing connection that you have already configured, or create a new one and then return to this task.
arn:aws:codeconnections:
or
Connect to GitHub
Repository name
Choose a repository in your GitHub account.
M-Yassir/weather-app
Cannot read properties of undefined (reading 'OwnerId') (Click here to retry)
You can type or paste the group path to any project that the provided credentials can access. Use the format 'group/subgroup/project'.
Default branch
Default branch will be used only when pipeline execution starts from a different source or manually started.
main
main
Choose the output artifact format.
CodePipeline default
AWS CodePipeline uses the default zip format for artifacts in the pipeline. Does not include Git metadata about the repository.
Full clone
AWS CodePipeline passes metadata about the repository that allows subsequent actions to do a full Git clone. Only supported for AWS CodeBuild actions. Learn more
Enable automatic retry on stage failure
Webhook events
Webhook - optional
Start your pipeline on push and pull request events.
Webhook event filters - optional
Starts your pipeline on a specific event.
Remove filters
Cancel
Previous
Next


Developer Tools > CodePipeline > Pipelines > Create new pipeline
Step 1
Choose creation option
Step 2
Choose pipeline settings
Step 3
Add source stage
Step 4
Add build stage
Step 5
Add test stage
Step 6
Add deploy stage
Step 7
Review
Add build stage Info
Step 4 of 7
Build - optional
Build provider
Choose the tool you want to use to run build commands and specify artifacts for your build action.
Commands
Other build providers
AWS CodeBuild
Project name
Choose a build project that you have already created in the AWS CodeBuild console. Or create a build project in the AWS CodeBuild console and then return to this task.
weather-app-project
or
Create project
Define buildspec override - optional
Buildspec file or definition that overrides the latest one defined in the build project, for this build only.
Environment variables - optional
Choose the key, value, and type for your CodeBuild environment variables. In the value field, you can reference variables generated by CodePipeline. Learn more
Name
Value
AWS_DEFAULT_REGION
your default region
Type
Plaintext
Remove
Name
Value
AWS_ACCOUNT_ID
your account id
Type
Plaintext
Remove
Name
Value
REPOSITORY_NAME
weather-app
Type
Plaintext
Remove
Name
Value
CONTAINER_NAME
weather-app-container
Type
Plaintext
Remove
Add environment variable
Build type
Single build
Triggers a single build.
Batch build
Triggers multiple builds as a single execution.
Region
United States (N. Virginia)
Input artifacts
Choose an input artifact for this action. Learn more
SourceArtifact
Defined by: Source
Enable automatic retry on stage failure
Cancel
Previous
Skip build stage
Next





On the review page (step 7), we click on create pipeline.
After waiting a little bit, the pipeline should be deployed successfully, we should get something like this:



When we modify the app code and rerun the pipeline, it will automatically rebuild the Docker image, push it to ECR, and deploy the updated version to ECS. The ALB seamlessly shifts traffic to the new containers once they pass health checks, ensuring a smooth rollout without downtime. This end-to-end automation keeps our app updated with every code change.

To see a demo of this, go watch the video in the repo and don't forget to delete all of these resources (ECS cluster, ALB, task definitions, pipeline…) when you finish.

Conclusion

This initiative represented a comprehensive exploration of containerized CI/CD implementation using AWS CodePipeline. Through strategic integration of ECS, ALB, and AWS developer tools, I engineered a resilient deployment pipeline that delivered scalability while operating within rigorous Free Tier limitations.

Key challenges & technical solutions:

1) Resource Allocation Optimization

The project presented unique constraints in resource management. Initial configurations using 1 vCPU/1GB task definitions proved unsustainable, while undersized 0.1 vCPU allocations resulted in performance degradation.
Solution: Conducted methodical capacity testing to identify the optimal 0.3 vCPU/0.3GB configuration, demonstrating that precise resource calibration can achieve stability even under tight constraints.

2) Port Mapping and Load Balancer Configuration

Architecting efficient traffic distribution required solving multiple integration challenges. The initial approach using static port mapping (hostPort: 80) created scaling limitations, while ALB target group misconfigurations disrupted service health monitoring.
Solution: Implemented dynamic port allocation (hostPort: 0) and refined path-based routing rules, establishing a foundation for elastic workload distribution.

3) Pipeline Automation and Permission Management

The deployment process uncovered critical automation gaps, particularly the missing imagedefinitions.json artifact, while permission constraints hindered CodeBuild operations.
Solution: Restructured the build specification to ensure proper artifact generation and implemented granular IAM policies, achieving both functionality and security compliance.

Professional development outcomes:

This engagement yielded significant technical competencies:

✓ Designed and validated cost-efficient ECS architectures

✓ Engineered auto-scaling solutions through dynamic load balancing

✓ Gained practical experience with AWS IAM policy troubleshooting

✓ Learned container optimization techniques for resource-constrained environments

✓ Developed systematic debugging skills for CI/CD pipelines

✓ Developed problem-solving skills for cloud deployment challenges

Strategic value & future applications:

The project not only demonstrated the effectiveness of AWS CI/CD services but also established a framework for infrastructure optimization. These learnings create opportunities to advance deployment methodologies through:

-Blue/green deployment patterns
-Infrastructure-as-Code adoption
-Kubernetes ecosystem integration