Proyecto Final

AquaSenseCloud

Infraestructura para la

Computación de Altas Prestaciones

Curso: 2024/2025

Rubén Gil Martínez

Guillermo López Pérez

ÍNDICE

[Cronograma](#_Toc178625155)

Diagrama [Arquitectura de la solución](#_Toc178625156)

[Listado de Recursos y Servicios con su funcionalidad](#_Toc178625155)

[Razonamiento de las decisiones de diseño(Solución y Automatización)](#_Toc178625156)

[Ejemplos de demostración de correcta funcionalidad](#_Toc178625156)

[Anexo de Replicación](#_Toc178625155)

[Anexo de Automatización 3](#_Toc178625155)

# Cronograma

| **Fecha** | **Tarea** | **Responsable** | **Duración** |
| --- | --- | --- | --- |
| 23 diciembre | Revisión inicial de requisitos, planificación general y asignación de tareas. | Ambos | 1 día |
| 24y26 diciembre | Diseño del pipeline de datos con la configuración de servicios de AWS (S3, Lambda, DynamoDB, SNS). | Ambos | 2 días |
| 27 diciembre | Implementación y pruebas del pipeline de datos. | Guillermo López | 1 día |
| 28-30 diciembre | Desarrollo del servidor web (aquasense.py) y dockerización (Dockerfile, requirements.txt). | Rubén Gil | 3 días |
| 31 diciembre | Pruebas locales del servidor y subida de la imagen del contenedor a AWS ECR. | Rubén Gil | 1 día |
| 2 enero | Despliegue de la infraestructura de red con CloudFormation (VPC, subredes, IGW, …). | Guillermo López | 1 día |
| 3-7 enero | Despliegue de la infraestructura de alto nivel con CloudFormation (Clúster, ALB, Servicio, …). | Ambos | 5 días |
| 8-10 enero | Pruebas finales de: Escalabilidad, tolerancia a fallos y correcto funcionamiento del servicio.  Preparación del informe. | Ambos | 3 días |

# Diagrama Arquitectura de la solución

Escala de tiempo

Descripción generada automáticamente con confianza media

# Listado de Recursos y Servicios con su funcionalidad

**1. Ingesta y Almacenamiento Crudo**

* **Amazon S3 (AWS::S3::Bucket)**: Almacena los archivos CSV de datos brutos con las medias y d. típicas de temperaturas, se utilizan como entrada para el pipeline de datos. La ingesta de nuevos datos provoca el desencadenamiento de la función Lambda desarrollada, por lo tanto, nuestro bucket es asignado como desencadenador.

**2. Pipeline de Datos**

* **AWS Lambda (AWS::Lambda::Function)**: Ejecuta funciones que procesan los archivos almacenados en S3, transformando dichos datos y cargándolos en una tabla de DynamoDB. Esos datos almacenados en la tabla ya están preparados para ser consultados.
* **Amazon SNS (AWS::SNS::Topic)**: Se utiliza para enviar notificaciones cuando la desviación estándar semanal de las temperaturas supera el umbral de 0.5. Esta notificación se publica en el tema SNS, informando a los suscriptores (en este caso los analistas) sobre la situación crítica en el monitoreo de temperaturas.

**3. Almacenamiento de Datos Procesados/Transformados**

* **DynamoDB (AWS::DynamoDB::Table)**: Almacena todos los datos transformados y procesados, listos para ser consultados, proporcionando así un acceso rápido y escalable a los datos.

**4. Recursos para el Desarrollo y Pruebas Iniciales**

* **Instancia EC2 (AWS::EC2::Instance):**  
  Utilizada para desarrollar, probar y ajustar el contenedor con la aplicación AquaSense. En esta instancia se configuró el entorno de Docker y se realizaron pruebas locales antes de enviar la imagen al repositorio de ECR.

**5. Infraestructura de Red y Conectividad**

* **VPC (AWS::EC2::VPC)**: Proporciona un entorno de red aislado donde se despliegan los recursos de la infraestructura.
* **Subnets (AWS::EC2::Subnet)**: Crea subredes públicas que permiten el acceso a Internet para los recursos dentro de la VPC.
* **Internet Gateway (AWS::EC2::InternetGateway)**: Facilita la conexión de la VPC a Internet, permitiendo que los analistas accedan a los servicios configurados y desplegados.
* **Route Table (AWS::EC2::RouteTable)**: Define las rutas de tráfico de red dentro de la VPC, asegurando que el tráfico fluya adecuadamente entre subredes y el Internet Gateway.

**6. Almacenamiento de Imagenes**

* **ECR (Amazon Elastic Container Registry)**: Repositorio para almacenar y versionar la imagen del contenedor de AquaSense después de desarrollarla en EC2. Este recurso nos facilita la integración con ECS para el despliegue en producción de nuestra aplicación.

**7. Recursos para el Despliegue de la Aplicación**

* **ECSCluster(AWS::ECS::Cluster):** Agrupa, administra y organiza todas las tareas y servicios relacionados con la ejecución del contenedor de la aplicación. Actúa como punto central de administración para la ejecución de tareas, asegurando que estas puedan desplegarse de manera distribuida y eficiente.
* **ALB(AWS::ElasticLoadBalancingV2::LoadBalancer):** Balanceador de carga que gestiona/balancea el tráfico de los analistas hacia las tareas ECS. Redirige las solicitudes desde el puerto 80(tráfico HTTP) al puerto 5000 del contenedor, además, mejora la disponibilidad y la tolerancia a fallos de nuestra aplicación.
* **ALBListener(AWS::ElasticLoadBalancingV2::Listener):** Configura reglas en el balanceador de carga para redirigir las solicitudes hacia el grupo de destino, escucha el tráfico de los analistas a través del puerto 80.
* **ECSTargetGroup(AWS::ElasticLoadBalancingV2::TargetGroup):** Asocia las tareas ECS con el balanceador de carga y configura verificaciones de estado mediante el endpoint /health que ha sido configurado en nuestra aplicación. Este permite conocer el estado en que se encuentra las tareas para que, en caso de que alguna falle, se marca la tarea como no saludable y eso informa al ECS Service de que hay que lanzar nuevas tareas para mantener su número deseado.

**8. Definición y Gestión de Tareas**

* **ECSTaskDefinition(AWS::ECS::TaskDefinition):** Este recurso describe las configuraciones necesarias para ejecutar contenedores en ECS. Especifica detalles como la imagen del contenedor a usar , la cantidad de CPU y memoria asignada y la compatibilidad con el modo Fargate, que permite ejecutar contenedores sin gestionar servidores subyacentes. Además, define un mapeo de puertos, asignando el puerto interno del contenedor (5000) al tráfico entrante. Es el punto central de la especificación para las tareas ECS.
* **ECSService(AWS::ECS::Service):**  
  Administra la ejecución de las tareas ECS, asegura la ejecución continua de un número deseado de tareas definidas en la configuración de tarea. En este caso, se garantiza que siempre haya dos tareas activas para manejar las solicitudes entrantes. Se utiliza el modo de lanzamiento Fargate para simplificar la gestión de la infraestructura. El servicio monitorea las tareas y las reinicia automáticamente si fallan, garantizando alta disponibilidad y resiliencia para nuestra aplicación.

**9. Escalado Automático**

* **ECS Scalable Target(AWS::ApplicationAutoScaling::ScalableTarget):** Define el rango de capacidad del servicio ECS, asegurando que el número de tareas FARGATE activas esté entre un mínimo de 2 y un máximo de 10. Este recurso actúa como el objetivo escalable que la política de escalado monitorea y ajusta según las métricas configuradas. Está asociado al recurso ECSService y permite mantener un nivel óptimo de disponibilidad y desempeño de nuestra aplicación.
* **Scaling Policy (High CPU)(AWS::ApplicationAutoScaling::ScalingPolicy):** Implementa una política de escalado automático que aumenta la cantidad de tareas activas cuando la utilización promedio de CPU en el clúster ECS supera el 75%. Esto asegura que la capacidad del servicio crezca dinámicamente para manejar un aumento en la carga de trabajo. Incluye un tiempo de enfriamiento (cooldown) de 60 segundos para evitar ajustes excesivamente frecuentes.
* **Scaling Policy (Low CPU) (AWS::ApplicationAutoScaling::ScalingPolicy):** Establece una política de reducción de escala que disminuye el número de tareas activas cuando el uso de CPU cae por debajo del 25%. Esto permite optimizar costos al liberar recursos durante períodos de baja demanda, manteniendo solo las tareas necesarias para soportar la carga en ese momento. Similar a la política de escalado ascendente, aplica un enfriamiento de 60 segundos.

**10. Seguridad**

* **ALB Security Group (AWS::EC2::SecurityGroup)**: Permite el acceso al balanceador de carga desde Internet(a través del puerto 80-HTTP), asegurando que el tráfico pueda alcanzar la aplicación.
* **Task Security Group (AWS::EC2::SecurityGroup)**: Permite el tráfico del ALB a las tareas de ECS(a través del puerto 5000), asegurando que las solicitudes lleguen al contenedor de la aplicación de manera distribuida y segura.

**Gestión e Implementación**

* **CloudFormation**: Automatiza el despliegue de la infraestructura mediante plantillas, asegurando la coherencia y la reproducibilidad en la implementación.
* **AWS CLI**: Utilizada para el desarrollo de la imagen y autenticación y gestión de recursos a la hora de subir la imagen del contenedor de nuestra aplicación a AWS ECR.
* **Docker**: Facilita el desarrollo y la contenedorización de la aplicación, asegurando un entorno de ejecución consistente.
* **IAM Roles (AWS::IAM::Role)**: Proporcionan los permisos necesarios para que las tareas de ECS accedan a DynamoDB y otros recursos de AWS. Permite que los distintos recursos de la arquitectura desplegada se puedan comunicar entre sí con seguridad.

# Explicación clara y concisa de los pasos seguidos para implementar la Solución y Automatización

**1) Ingesta de Datos y Pipeline**

Iniciamos creando una tabla de DynamoDB estableciendo como clave el atributo fecha, aquí serán almacenados los datos transformados para su posterior consulta por los analistas a través de nuestra aplicación. Después, creamos un tópico en AWS SNS porque se nos pide informar cuando la desviación típica de la temperatura de una semana supere el valor 0.5, a este tópico asociamos como suscriptores a los analistas encargados de recibir las alertas y asociamos como productor a la función AWS Lambda que procesará los datos e identificará aquellas desviaciones típicas que provoquen la alerta.

Definimos un bucket mediante AWS S3 para almacenar los datos de las temperaturas del Mar Menor en crudo como objetos, tras esto definimos la función AWS Lambda antes comentada, esta procesará los datos de estos ficheros que se suban al bucket de manera automática ya que definimos la subida de datos al bucket como un desencadenador de la función. AWS Lambda procesa y transforma los datos de manera que los deja preparados para su consulta y los almacena en la tabla creada de Dynamo DB.

**2) Desarrollo y Pruebas de la aplicación mediante un contenedor de Docker**

Una vez realizado todo el Pipeline de datos pasamos a desarrollar nuestra aplicación, para ello desplegamos una pila de CloudFormation con la infraestructura necesaria para darle soporte a una instancia EC2 donde desarrollaremos nuestra aplicación mediante un contenedor de Docker.

A la instancia EC2 accedemos mediante SSH y dentro de ella desarrollamos toda la contenedorización. Creamos un entorno(carpeta) del contenedor donde generamos los ficheros (requirements.txt, Dockerfile y aquasense.py) necesarios para el lanzamiento de nuestro contenedor, comentar que en nuestra aplicación simplemente se definen una serie de endpoints donde los analistas llamarán dependiendo del tipo de dato que necesiten y la fecha exacta que quieran contemplar, nuestra aplicación simplemente accede a esos datos almacenados en nuestra tabla de Dynamo DB utilizando el SDK de AWS (boto3) y los devuelve al analista que lo necesite. Una vez definidos estos ficheros, se lanza el contenedor y se realizan una serie de pruebas locales para comprobar su correcto funcionamiento. Una vez asegurado el correcto funcionamiento de nuestra aplicación, pasamos a crear un repositorio ECR en AWS, este repositorio nos servirá para almacenar la imagen de nuestro contenedor en AWS y así poder ser utilizada a la hora de desplegar nuestro servicio. Con el repositorio ECR creado, subimos la imagen del contenedor a AWS ECR y así ya podemos usar su URI para las tareas de la infraestructura final.

**3) Automatización del aprovisionamiento de la infraestructura de red**

Desplegamos la plantilla de CloudFormation, que será la infraestructura de red, y que soportará los recursos de la solución final, se define una VPC con dos subredes públicas y un Internet Gateway para permitir la comunicación con el exterior.

**4) Automatización del despliegue de la infraestructura de la solución final (ECS)**

Una vez aprovisionada la infraestructura de red pasamos a lanzar la plantilla de CloudFormation que automatiza la implementación de la infraestructura final de la solución.

Creamos un clúster que contendrá las instancias FARGATE, seguido de definir un servicio que gestionará los contenedores con un mínimo de dos réplicas usando las subredes ya configuradas en la anterior pila, tener en cuenta que las tareas se despliegan de manera balanceada entre las dos subredes.

Este servicio tendrá dos políticas de escalado horizontal usando la métrica de utilización media de la CPU, simplemente se añadirá una tarea si se supera el 75% de utilización media y se eliminará una tarea (en caso de que no estemos en el número deseado de tareas) cuando haya una utilización menor del 25%. Es necesaria la creación de la definición de tarea que escuchará en el puerto 5000, estas tareas son soportadas por instancias FARGATE.

También es necesario crear un balanceador de carga junto a su agente de escucha y a su grupo de destino que serán los encargados de redirigir el tráfico de los analistas hacia las tareas. Por otro lado comentar la importancia de crear los grupos de seguridad, tanto para las tareas como para el balanceador de carga, al balanceador de carga restringimos el tráfico solamente a solicitudes HTTP y el tráfico hacia nuestras tareas estará restringido para que solo se pueda acceder a través del balanceador de carga.

# Ejemplos de demostración de correcta funcionalidad

* ***Función Lambda:***

***Captura de pantalla de computadora

Descripción generada automáticamente***

*Ejemplo contenido del fichero: ‘temperatures\_ingesta1*

*Texto

Descripción generada automáticamente*

*Almacenamiento datos transformados en DynamoDB*

*Captura de pantalla de computadora

Descripción generada automáticamente*

*Pantalla de un video juego

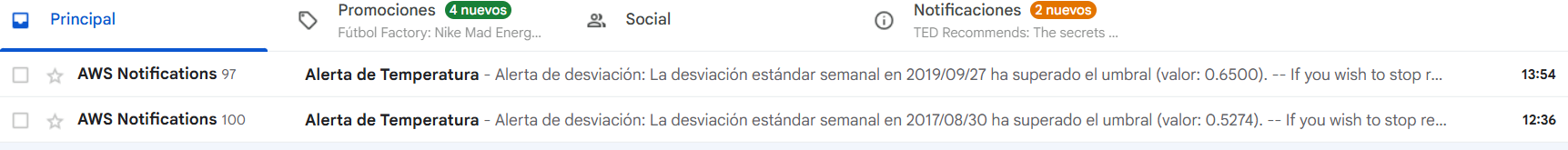
Descripción generada automáticamente con confianza mediaPantalla de un video juego

Descripción generada automáticamente con confianza media*

***#NOTA:*** *Observar que la función lambda trata con la posibilidad de ingesta de datos sin importar los siguientes factores: su orden temporal, su repetición y datos mensuales separados en distintos ficheros.*

***Comentar también que los datos transformados se almacenan en la tabla de DynamoDB correctamente si se realiza la ingesta de manera controlada y con ficheros de un tamaño medio(2-3 Kbytes) como los utilizados en las pruebas realizadas.***

*Envío del correo a los analistas mediante un tópico definido mediante AWS SNS*

**

* ***ECR:***

***Captura de pantalla de un celular

Descripción generada automáticamente***

* ***Automatización del Aprovisionamiento de la Infraestructura mediante CloudFormation:***

***Captura de pantalla de computadora

Descripción generada automáticamente con confianza media***

***Captura de pantalla de un celular

Descripción generada automáticamente***

* ***Auto Escalado con métricas de utilización de recursos:***

***#NOTA:*** *Para probar el correcto funcionamiento del Auto Escalado de nuestro servicio definimos un programa que lanzase miles de peticiones por segundo a nuestra aplicación. Aquí se observan las métricas recabadas durante ese periodo y las decisiones tomadas por ECS Service gracias al auto escalado definido:*

*Métrica Utilización de CPU*

***Captura de pantalla de un videojuego

Descripción generada automáticamente***

*Métrica Utilización de Memoria*

*Captura de pantalla de un videojuego

Descripción generada automáticamente*

*#NOTA: Observar que la utilización de la memoria es muy baja en nuestra aplicación en comparación con la CPU, por ello, la métrica que utilizamos en el auto escalado del servicio es la utilización media de la CPU por parte de las tareas FARGATE desplegadas.*

*Auto Escalado del Servicio (El número de tareas deseadas inicial es 2)*

*Captura de pantalla de computadora

Descripción generada automáticamente*

* ***Funcionamiento De la Aplicación:***

*Distintas consultas que pueden realizar los analistas a través del servidor web desplegado por las tareas. En la propia URL definen el tipo de dato a consultar con su mes y año específico, veamos también el correcto funcionamiento del endpoint(/health).*

***(/maxdiff)***

***Captura de pantalla de un celular

Descripción generada automáticamente***

***(/sd)***

***Captura de pantalla de computadora

Descripción generada automáticamente***

***(/temp)***

***Captura de pantalla de un celular

Descripción generada automáticamente***

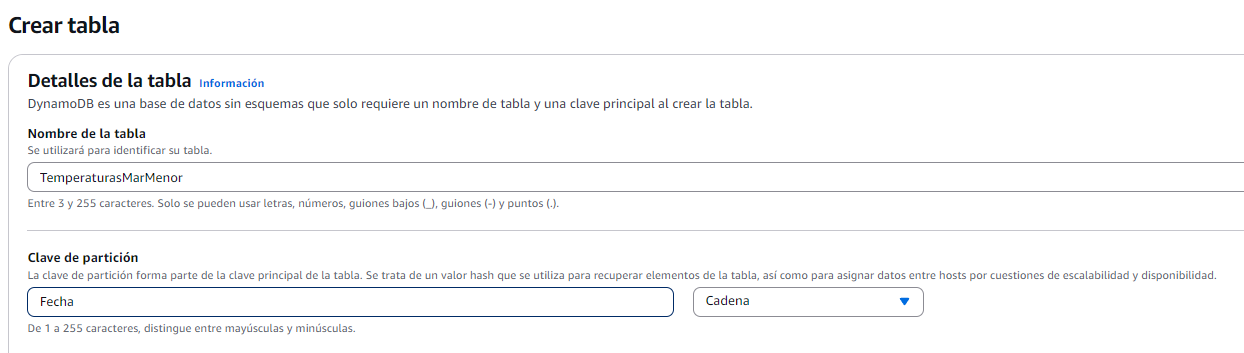
***(/health)***

***Captura de pantalla de un celular

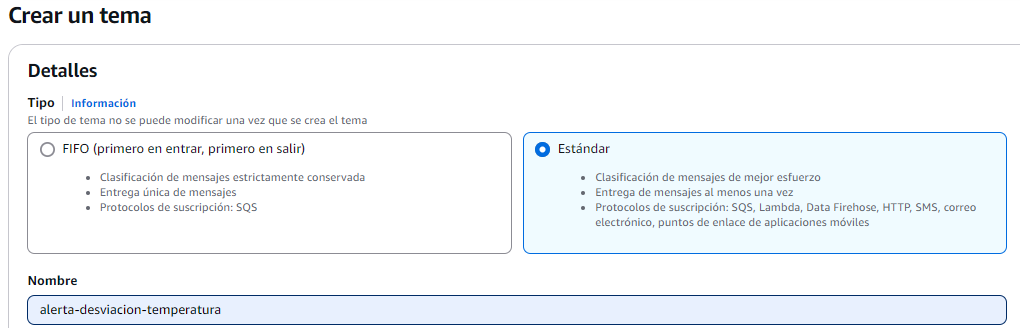
Descripción generada automáticamente***

# Anexo de Replicación

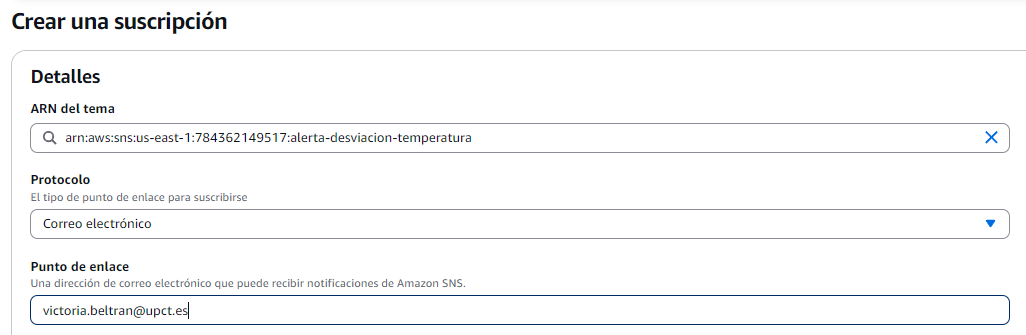
Creación de tabla DynamoDB con nombre TemperaturasMarMenor y clave de partición Fecha.



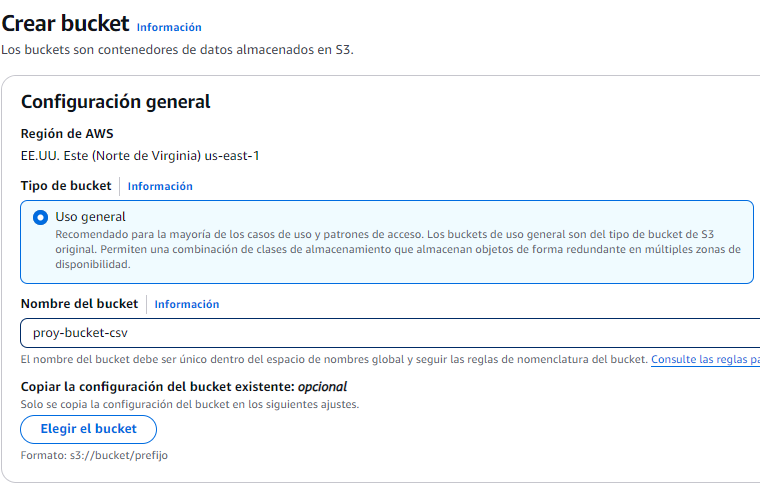
Creación de un Tema (Amazon SNS) tipo estándar con nombre alerta-desviacion-temperatura.



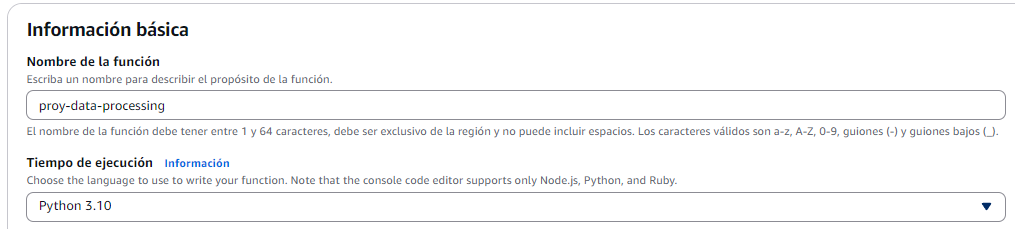
Creamos una suscripción dentro del tema creado con protocolo: correo electrónico y punto de enlace: victoria.beltran@upct.es.



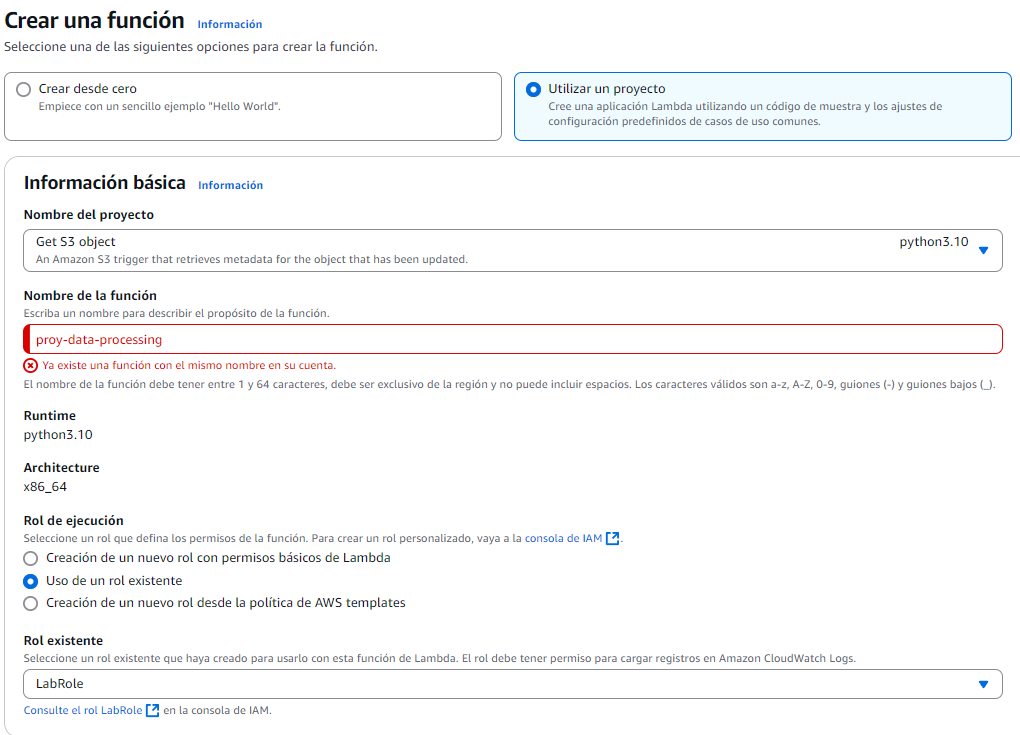
Creamos un bucket con nombre proy-bucket-csv.



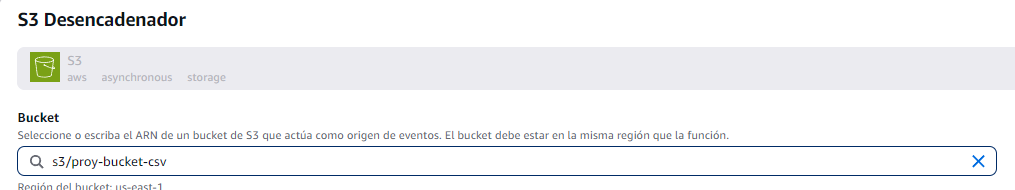
Creamos una función lambda con nombre proy-data-processing y tiempo de ejecución Python 3.10.



Creamos una función lambda. Utilizar un proyecto, nombre proy-data-processing, Rol existente: LabRole.



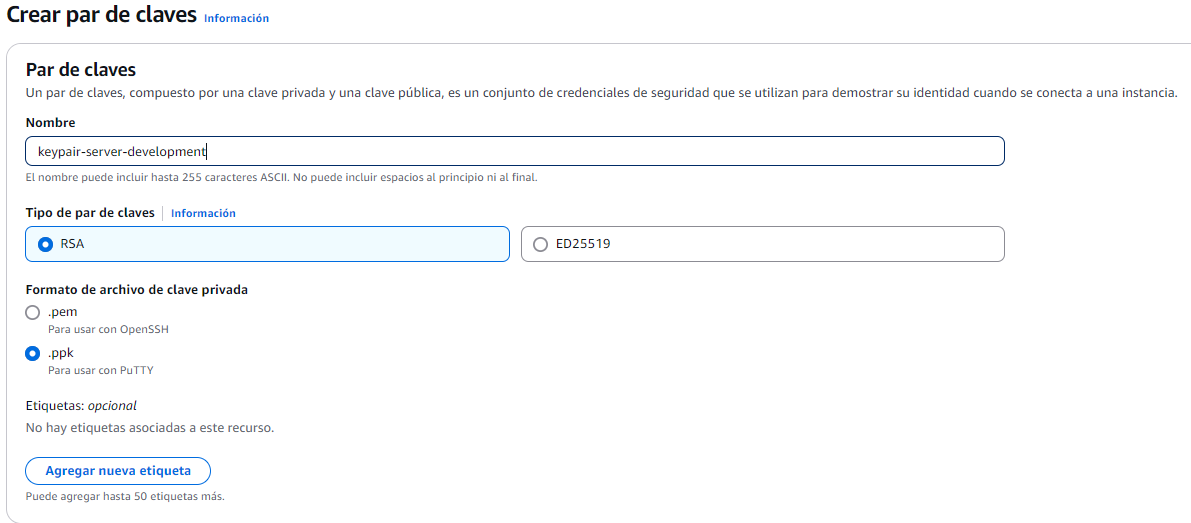
Aún en la creación de la función, elegimos el bucket creado anteriormente (proy-bucket-csv).



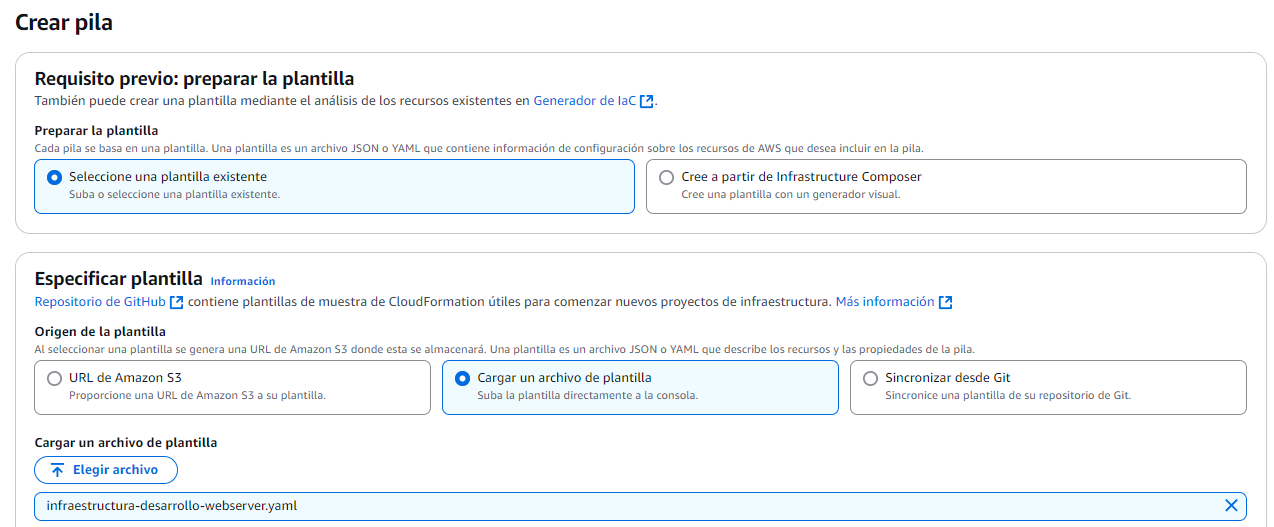
En el código de la función lambda pegamos el contenido del script función\_lambda.py.

Para que funcione correctamente se deberá utilizar el arn del tópico creado (único de cada cuenta). Así que hay que modificarlo.

Creamos un par de claves con el nombre keypair-server-development, RSA, formato ppk.

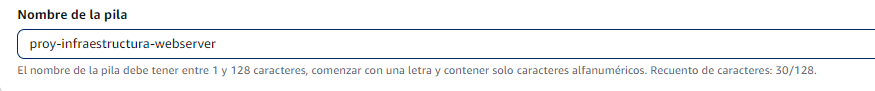


Ahora vamos a CloudFormation y crearemos una pila con recursos nuevos.

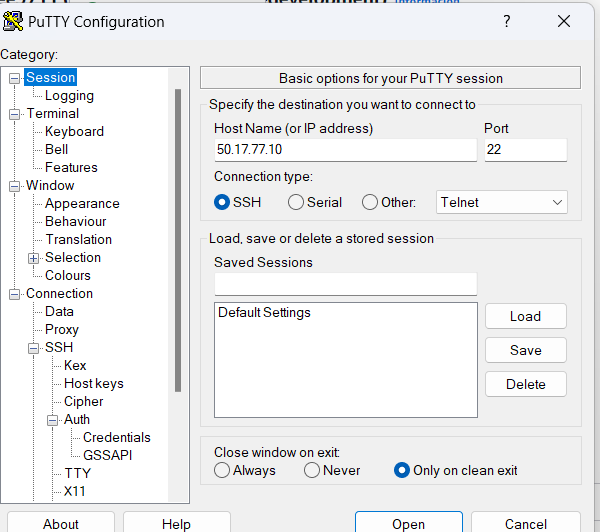


Cargamos el fichero infraestructura-desarrollo-webserver.yaml.

Nombre de la pila: proy-infraestructura-webserver. Seleccionamos el LabRole y la creamos.

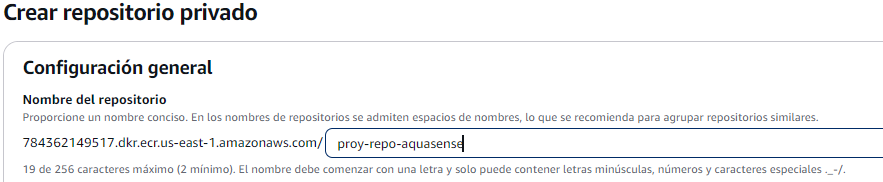


Nos conectamos con a la IP pública de la instancia EC2 creada a través del cliente PuTTY, utilizando el par de claves creado anteriormente.



Una vez dentro, ejecutar en orden los comandos del archivo AWS-CLI-Contenedorizacion-ECR.txt (Pasos 1 y 2) Los contenidos de Dockerfile, requirements.txt y aquasense.py estarán disponibles en la entrega.

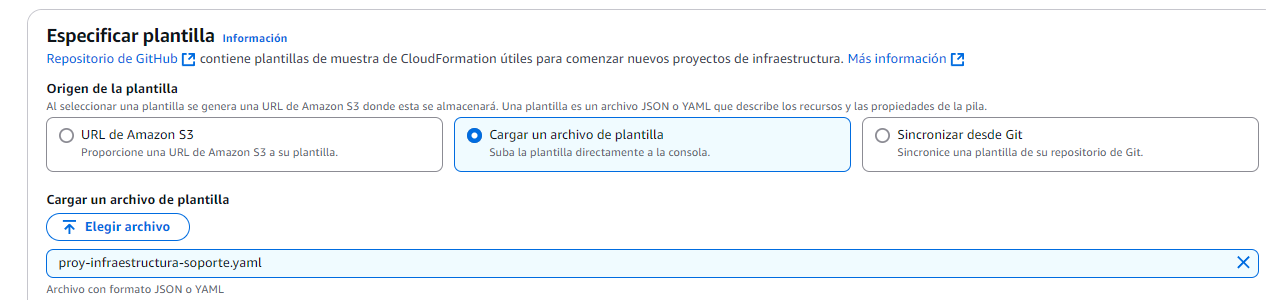
Una vez hecho esto, vamos a crear un repositorio llamado proy-repo-aquasense.



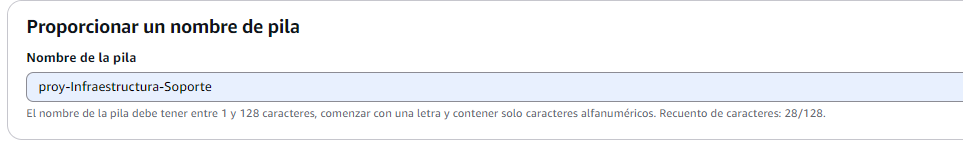
Una vez creado volvemos a la instancia y ejecutamos los comandos del archivo AWS-CLI-Contenedorizacion-ECR.txt (Paso 3). Tanto para la autenticación como para agregar la copia al repositorio se necesita el id de la cuenta (único).

Ahora nos vamos a CloudFormation para crear la primera pila de la infraestructura de la solución final.

Creamos la pila cargando el archivo proy-infraestructura-soporte.yaml.

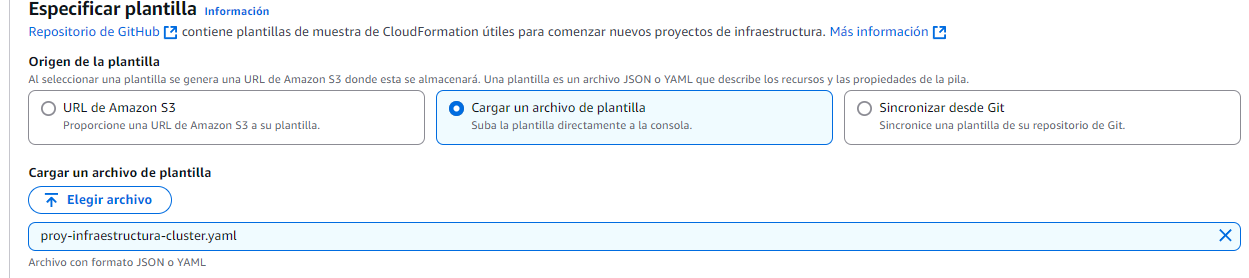


Le damos el nombre de proy-Infraestructura-Soporte.

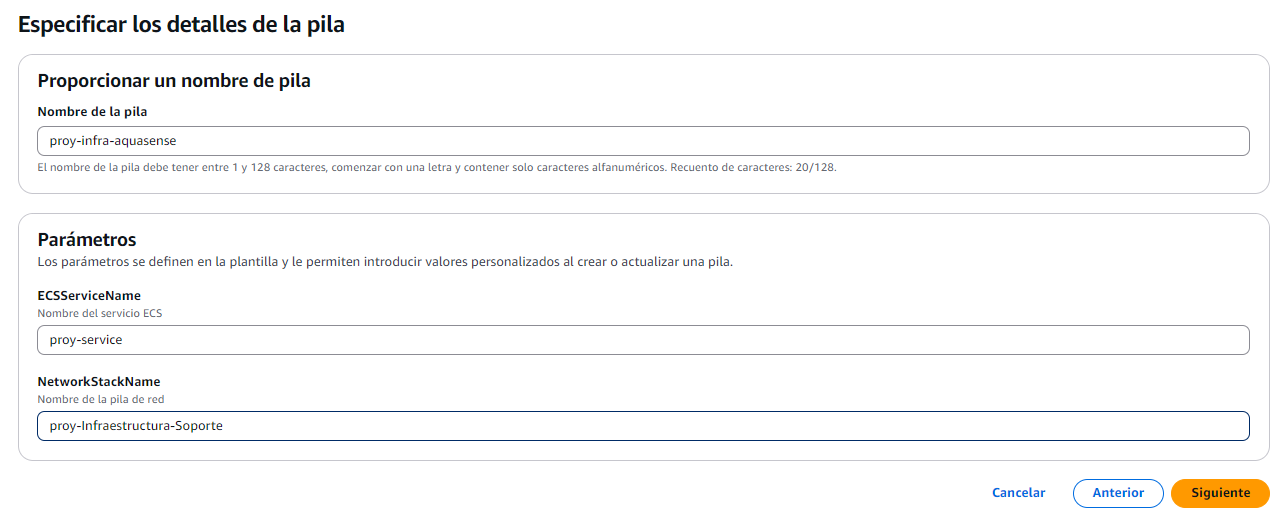


Seleccionamos el LabRole para los permisos, y la creamos.

Lo mismo para la pila de la infraestructura. Seleccionamos el archivo proy-infraestructura-cluster.yaml.



Nombre de la pila: proy-infra-aquasense. Parámetros ECSServiceName: proy-service. NetworkStackName: proy-Infraestructura-Soporte.



Seleccionamos el LabRole para los permisos y la creamos.

Hecho esto, ya tenemos toda la infraestructura desplegada.

# Anexo de Automatización

***1)Plantilla de CloudFormation infraestructura-desarrollo-webserver.yaml:***

AWSTemplateFormatVersion: 2010-09-09

Description: >-

  Despliegue de la infraestructura de red, grupos de seguridad y una instancia EC2 para desarrollo y pruebas de correcto funcionamiento de nuestra aplicacion(servidor web)

Resources:

  ## VPC

  VPC:

    Type: AWS::EC2::VPC

    Properties:

      EnableDnsSupport: true

      EnableDnsHostnames: true

      CidrBlock: 192.168.0.0/16

      Tags:

        - Key: Name

          Value: vpc-server-development

  ## Internet Gateway

  InternetGateway:

    Type: AWS::EC2::InternetGateway

    Properties:

      Tags:

        - Key: Name

          Value: Internet-Gateway-server-development

  VPCGatewayAttachment:

    Type: AWS::EC2::VPCGatewayAttachment

    Properties:

      VpcId: !Ref VPC

      InternetGatewayId: !Ref InternetGateway

  ## Public Route Table

  PublicRouteTable:

    Type: AWS::EC2::RouteTable

    Properties:

      VpcId: !Ref VPC

  PublicRoute:

    Type: AWS::EC2::Route

    DependsOn: VPCGatewayAttachment

    Properties:

      RouteTableId: !Ref PublicRouteTable

      DestinationCidrBlock: 0.0.0.0/0

      GatewayId: !Ref InternetGateway

  ## Public Subnet

  PublicSubnet:

    Type: AWS::EC2::Subnet

    Properties:

      VpcId: !Ref VPC

      CidrBlock: 192.168.1.0/24

      AvailabilityZone: us-east-1a

      Tags:

        - Key: Name

          Value: public-subnet-server-development

  PublicSubnetRouteTableAssociation:

    Type: AWS::EC2::SubnetRouteTableAssociation

    Properties:

      SubnetId: !Ref PublicSubnet

      RouteTableId: !Ref PublicRouteTable

  # Docker instance

  DockerInstance:

    Type: AWS::EC2::Instance

    Properties:

      InstanceType: t2.micro

      ImageId: ami-08a0d1e16fc3f61ea

      KeyName: keypair-server-development

      IamInstanceProfile: LabInstanceProfile

      NetworkInterfaces:

        - GroupSet:

            - !Ref FlaskSecurityGroup

          AssociatePublicIpAddress: true

          DeviceIndex: 0

          DeleteOnTermination: true

          SubnetId: !Ref PublicSubnet

      Tags:

        - Key: Name

          Value: DockerInstance-server-development

      UserData:

        Fn::Base64: !Sub |

          #!/bin/bash

          dnf update -y

          #install Docker

          dnf -y install docker

  DiskVolume:

    Type: AWS::EC2::Volume

    Properties:

      Size: 100

      AvailabilityZone: !GetAtt DockerInstance.AvailabilityZone

    DeletionPolicy: Snapshot

  DiskMountPoint:

    Type: AWS::EC2::VolumeAttachment

    Properties:

      InstanceId: !Ref DockerInstance

      VolumeId: !Ref DiskVolume

      Device: /dev/sdh

  FlaskSecurityGroup:

    Type: AWS::EC2::SecurityGroup

    Properties:

      GroupDescription: Permite acceso SSH al servidor Flask a traves del puerto 22 y acceso a traves del puerto 5000 para comprobar el correcto funcionamiento del servidor

      VpcId: !Ref VPC

      SecurityGroupIngress:

        - IpProtocol: tcp

          FromPort: 5000

          ToPort: 5000

          CidrIp: 0.0.0.0/0

        - IpProtocol: tcp

          FromPort: 22

          ToPort: 22

          CidrIp: 0.0.0.0/0

      Tags:

        - Key: Name

          Value: FlaskSG-server-development

***2)Plantilla de CloudFormation proy-infraestructura-soporte.yaml:***

AWSTemplateFormatVersion: 2010-09-09

Description: >-

  Despliegue de la infraestructura de red y grupos de seguridad de soporte para el sistema que alojara nuestro cluster de instancias ec2 con nuestra aplicacion de Aquasense.

######################

# Resources section

######################

Resources:

  ## VPC

  VPC:

    Type: AWS::EC2::VPC

    Properties:

      EnableDnsSupport: true

      EnableDnsHostnames: true

      CidrBlock: 192.168.0.0/16

      Tags:

        - Key: Name

          Value: proy-vpc

  ## Internet Gateway

  InternetGateway:

    Type: AWS::EC2::InternetGateway

    Properties:

      Tags:

        - Key: Name

          Value: proy-Internet-Gateway

  VPCGatewayAttachment:

    Type: AWS::EC2::VPCGatewayAttachment

    Properties:

      VpcId: !Ref VPC

      InternetGatewayId: !Ref InternetGateway

  ## Tabla de enrutamiento y ruta

  PublicRouteTable:

    Type: AWS::EC2::RouteTable

    Properties:

      VpcId: !Ref VPC

  PublicRoute:

    Type: AWS::EC2::Route

    DependsOn: VPCGatewayAttachment

    Properties:

      RouteTableId: !Ref PublicRouteTable

      DestinationCidrBlock: 0.0.0.0/0

      GatewayId: !Ref InternetGateway

  ## Subredes publicas

  PublicSubnet1:

    Type: AWS::EC2::Subnet

    Properties:

      VpcId: !Ref VPC

      CidrBlock: 192.168.1.0/24

      AvailabilityZone: us-east-1a

      Tags:

        - Key: Name

          Value: proy-public-subnet-zone1

  PublicSubnet2:

    Type: AWS::EC2::Subnet

    Properties:

      VpcId: !Ref VPC

      CidrBlock: 192.168.2.0/24

      AvailabilityZone: us-east-1b

      Tags:

        - Key: Name

          Value: proy-public-subnet-zone2

  # Tablas de enrutamiento de las subredes

  PublicSubnetRouteTableAssociation1:

    Type: AWS::EC2::SubnetRouteTableAssociation

    Properties:

      SubnetId: !Ref PublicSubnet1

      RouteTableId: !Ref PublicRouteTable

  PublicSubnetRouteTableAssociation2:

    Type: AWS::EC2::SubnetRouteTableAssociation

    Properties:

      SubnetId: !Ref PublicSubnet2

      RouteTableId: !Ref PublicRouteTable

Outputs:

  VPC:

    Description: VPC ID

    Value: !Ref VPC

    Export:

      Name: !Sub ${AWS::StackName}-VpcId

  PublicSubnet1:

    Description: Subred pública 1 ID

    Value: !Ref PublicSubnet1

    Export:

      Name: !Sub ${AWS::StackName}-PublicSubnet1Id

  PublicSubnet2:

    Description: Subred pública 2 ID

    Value: !Ref PublicSubnet2

    Export:

      Name: !Sub ${AWS::StackName}-PublicSubnet2Id

***3)Plantilla de CloudFormation proy-infraestructura-cluster.yaml:***

AWSTemplateFormatVersion: 2010-09-09

Description: >-

  Plantilla para desplegar un clúster ECS con balanceador de carga ALB que redirige el tráfico de los clientes desde el puerto 80 (HTTP) hasta el puerto 5000 de las

  tareas donde se encuentra el contenedor de la aplicación AquaSense.

Parameters:

  ECSServiceName:

    Type: String

    Description: Nombre del servicio ECS

  NetworkStackName:

    Type: String

    Description: Nombre de la pila de red

Resources:

  # Cluster ECS

  ECSCluster:

    Type: AWS::ECS::Cluster

    Properties:

      ClusterName: !Sub "${ECSServiceName}-Cluster"

  ALBSecurityGroup:

    Type: AWS::EC2::SecurityGroup

    Properties:

      GroupDescription: "Permitir acceso al balanceador de carga ALB"

      VpcId:

        Fn::ImportValue: !Sub ${NetworkStackName}-VpcId

      SecurityGroupIngress:

        - IpProtocol: tcp

          FromPort: 80

          ToPort: 80

          CidrIp: 0.0.0.0/0

  TaskSecurityGroup:

    Type: AWS::EC2::SecurityGroup

    Properties:

      GroupDescription: "Allow traffic from ALB to ECS tasks"

      VpcId:

        Fn::ImportValue: !Sub ${NetworkStackName}-VpcId

      SecurityGroupIngress:

        - IpProtocol: tcp

          FromPort: 5000

          ToPort: 5000

          SourceSecurityGroupId: !Ref ALBSecurityGroup

  # Balanceador de carga ALB

  ALB:

    Type: AWS::ElasticLoadBalancingV2::LoadBalancer

    Properties:

      Name: !Sub "${ECSServiceName}-ALB"

      IpAddressType: ipv4

      SecurityGroups:

        - !Ref ALBSecurityGroup

      Subnets:

        - Fn::ImportValue: !Sub ${NetworkStackName}-PublicSubnet1Id

        - Fn::ImportValue: !Sub ${NetworkStackName}-PublicSubnet2Id

  ECSLoadBalancerTargetGroup:

    Type: AWS::ElasticLoadBalancingV2::TargetGroup

    Properties:

      Name: !Sub "${ECSServiceName}-TargetGroup"

      Port: 5000

      Protocol: HTTP

      VpcId:

        Fn::ImportValue: !Sub ${NetworkStackName}-VpcId

      TargetType: ip

      HealthCheckPath: "/health"

      Matcher:

        HttpCode: "200"

  ALBListener:

    Type: AWS::ElasticLoadBalancingV2::Listener

    Properties:

      DefaultActions:

        - Type: "forward"

          TargetGroupArn: !Ref ECSLoadBalancerTargetGroup

      LoadBalancerArn: !Ref ALB

      Port: 80

      Protocol: HTTP

  # Definición de tarea ECS

  ECSTaskDefinition:

    Type: AWS::ECS::TaskDefinition

    Properties:

      Family: !Sub "${ECSServiceName}-Task"

      ExecutionRoleArn: !Sub "arn:aws:iam::${AWS::AccountId}:role/LabRole"

      TaskRoleArn: !Sub "arn:aws:iam::${AWS::AccountId}:role/LabRole"

      RequiresCompatibilities:

        - FARGATE

      Cpu: "256"

      Memory: "512"

      NetworkMode: awsvpc

      ContainerDefinitions:

        - Name: !Sub "${ECSServiceName}-Container"

          Image: !Sub "${AWS::AccountId}.dkr.ecr.${AWS::Region}.amazonaws.com/proy-repo-aquasense:latest"

          PortMappings:

            - ContainerPort: 5000

              Protocol: tcp

  # Servicio ECS

  ECSService:

    Type: AWS::ECS::Service

    DependsOn: ALB

    Properties:

      ServiceName: !Sub "${ECSServiceName}-Service"

      Cluster: !Ref ECSCluster

      TaskDefinition: !Ref ECSTaskDefinition

      DesiredCount: 2

      LaunchType: FARGATE

      NetworkConfiguration:

        AwsvpcConfiguration:

          AssignPublicIp: ENABLED

          Subnets:

            - Fn::ImportValue: !Sub ${NetworkStackName}-PublicSubnet1Id

            - Fn::ImportValue: !Sub ${NetworkStackName}-PublicSubnet2Id

          SecurityGroups:

            - !Ref TaskSecurityGroup

      LoadBalancers:

        - TargetGroupArn: !Ref ECSLoadBalancerTargetGroup

          ContainerName: !Sub "${ECSServiceName}-Container"

          ContainerPort: 5000

  # Escalado automático horizontal(Utilización de CPU)

  ECSServiceScalingTarget:

    DependsOn: ECSService

    Type: AWS::ApplicationAutoScaling::ScalableTarget

    Properties:

      MaxCapacity: 10

      MinCapacity: 2

      ResourceId: !Sub "service/${ECSCluster}/${ECSServiceName}-Service"

      RoleARN: !Sub "arn:aws:iam::${AWS::AccountId}:role/LabRole"

      ScalableDimension: ecs:service:DesiredCount

      ServiceNamespace: ecs

  ECSServiceScalingPolicyHighCPU:

    Type: AWS::ApplicationAutoScaling::ScalingPolicy

    Properties:

      PolicyName: !Sub "${ECSServiceName}-ScaleUp"

      PolicyType: TargetTrackingScaling

      ScalingTargetId: !Ref ECSServiceScalingTarget

      TargetTrackingScalingPolicyConfiguration:

        TargetValue: 75.0

        PredefinedMetricSpecification:

          PredefinedMetricType: ECSServiceAverageCPUUtilization

        ScaleOutCooldown: 60

        ScaleInCooldown: 60

  ECSServiceScalingPolicyLowCPU:

    Type: AWS::ApplicationAutoScaling::ScalingPolicy

    Properties:

      PolicyName: !Sub "${ECSServiceName}-ScaleDown"

      PolicyType: TargetTrackingScaling

      ScalingTargetId: !Ref ECSServiceScalingTarget

      TargetTrackingScalingPolicyConfiguration:

        TargetValue: 25.0

        PredefinedMetricSpecification:

          PredefinedMetricType: ECSServiceAverageCPUUtilization

        ScaleOutCooldown: 60

        ScaleInCooldown: 60

Outputs:

  LoadBalancerDNSName:

    Description: DNS del ALB

    Value: !GetAtt ALB.DNSName

***4) AWS CLI:***

**PASO 1)**

Crear el directorio web-server-development mediante "mkdir"

Abrir y guardar en ficheros de texto por separado el contenido de: Dockerfile, requirements.txt y aquasense.py, mediante "nano".

**PASO 2)**

### DESARROLLO DE LA IMAGEN QUE SE USARÁ PARA LANZAR EL CONTENEDOR ###

sudo systemctl start Docker

sudo docker build -t aquasense-container . # CONSTRUIR LA IMAGEN

sudo docker images # PARA COMPROBAR CORRECTA CREACIÓN IMAGEN

"Este comando ejecuta el contenedor y la aplicación dentro él escuchando en el puerto 5000. El primer 5000 enlaza el contenedor al puerto 5000 del

sistema y el segundo 5000 asigna el puerto 5000 del contenedor al puerto 5000 del sistema."

sudo docker run -p 5000:5000 -d aquasense-container # EJECUTAR EL CONTENEDOR

sudo docker ps # PARA COMPROBAR CORRECTA EJECUCIÓN DEL CONTENEDOR

**PASO 3)**

### SUBIDA DE LA IMAGEN DEL CONTENDOR DE NUESTRA APLICACIÓN A AWS ECR ###

## Primer comando de esta ventana para la autenticación en Amazon ECR.

1) aws ecr get-login-password --region us-east-1 | sudo docker login --username AWS --password-stdin 471112833695.dkr.ecr.us-east-1.amazonaws.com

## Realización de una copia local de la imagen del contenedor que contiene nuestra aplicación (servidor web):

2) sudo docker tag aquasense-container 471112833695.dkr.ecr.us-east-1.amazonaws.com/proy-repo-aquasense

## Envio de la copia de la imagen Docker a nuestro repositorio ECR en AWS:

3) sudo docker push 471112833695.dkr.ecr.us-east-1.amazonaws.com/proy-repo-aquasense