

Despliegue de un Clúster Docker Swarm en AWS con ALB, S3 y Bases de Datos

Brayan J. Acuña

Juan D. Ayala

German E. Ardila

Daniel E. Novoa

Víctor A. Noriega

Universidad Popular del Cesar

Facultad de Ingenierías y Tecnológicas, Ingeniería de Sistemas

Ing. Salomón Cadena de la Hoz

Valledupar, Colombia

08 de abril de 2025

Introducción

En la actualidad, las arquitecturas basadas en la nube y los contenedores se han convertido en una solución eficiente y escalable para el desarrollo y despliegue. Con la creciente demanda de disponibilidad, rendimiento y resiliencia, las organizaciones están adoptando tecnologías que les permitan automatizar sus infraestructuras.

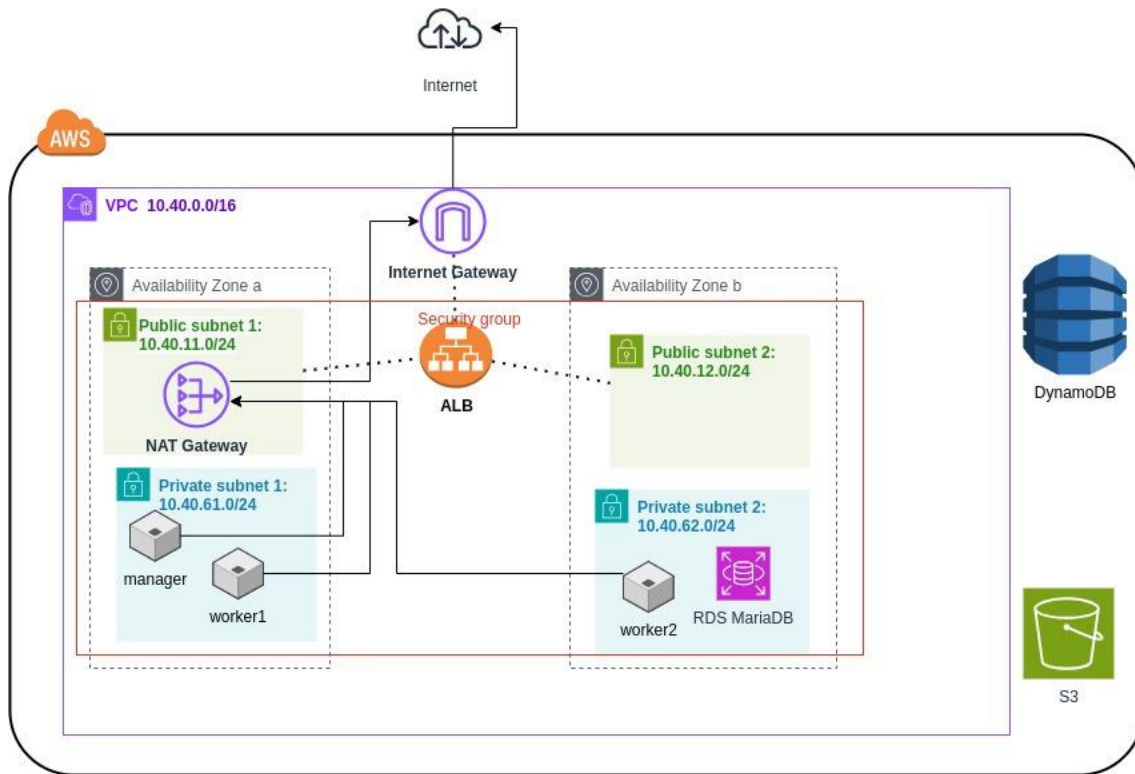
En este contexto **Amazon Web Services (AWS)** ofrece un ecosistema robusto de servicios que facilita la implementación de soluciones modernas, mientras que **Docker Swarn** permite la orquestación y gestión de contenedores de forma distribuida.

En este informe se documenta la implementación de una arquitectura Cloud utilizando AWS y Docker Swarn para desplegar una aplicación web con Frontend y Backend, integrando además servicios complementarios como almacenamiento como **Amazon RDS** y almacenamiento NoSQL con **Amazon DynamoDB**.

Objetivos

- Implementar un clúster de **Docker Swarn** en **AWS EC2**
- Desplegar una aplicación web (frontend y backend)
- Conectarla con **S3** y **bases de datos (RDS + DynamoDB)**

Arquitectura utilizada



La arquitectura implementada se basa en una solución altamente disponible y escalable utilizando servicios de **AWS**, juntos con un cluster de contenedores mediante **Docker Swarn**. Esta infraestructura fue diseñada para permitir el despliegue eficiente de una aplicación con Frontend y Backend asegurando disponibilidad, separación por capas y acceso seguro a los recursos.

Componente Principales

- **VPC (Virtual Private Cloud):** Se creó una red virtual personalizada con un rango de dirección IP privadas y públicas (255).
- **Subredes:**
 - **Públicas (10.40.11.0/24 y 10.40.12.0/24):** Distribuidas en dos zonas de disponibilidad, permiten la entrada de tráfico externo a través del **Internet Gateway** y alojan servicios como el **NAT Gateway**.

- **Privada (10.40.61.0/14 y 10.40.62.0/14):** Utilizadas para alojar los nodos del cluster (manager y workers), así como servicios internos como la base de datos en **Amazon RDS**.
- **Cluster Docker Swarn:** Compuesto por un nodo **Manager** y dos nodos **Worker**, desplegados en subredes probadas para mejorar la seguridad. El manager coordina la orquestación de servicios y tareas en los contenedores.
- **ALB (Application Load Balancer):** Configurado en las subredes públicas, permite distribuir el tráfico entrante de manera equitativa entre los servicios desplegados en los contenedores. Se conecta a **Target Groups** donde se registran las instancias con los servicios expuestos.
- **NAT Gateway:** Instalado en la subred publica para permitir que los recursos dentro de las subredes privadas (como los nodos del cluster) accedan a Internet para actualizaciones o descargas sin exponer sus direcciones IP públicas.
- **Amazon S3:** Utilizados para almacenamiento de objetos, como imágenes o archivos estáticos generados por el frontend o backend.
- **Amazon DynamoDB:** Base de datos NoSQL altamente escalable que puede ser utilizada para funcionalidades como almacenamiento de tokens, sesiones o catálogos rápidos.
- **Security Groups:** Definidos para controlar el trafico de red permitido hacia las instancias EC2, el ALB y los servicios internos. Se establecieron reglas específicas de entrada y salida según el rol del recurso.

Consideraciones de diseño:

Se utilizaron dos zonas de disponibilidad para asegurar tolerancia a fallos. Se mantuvieron servicios críticos como base de datos y clúster en subredes privadas, como acceso restringido a través del NAT Gateway. El uso de ALB y Docker permite

escalar los servicios de la aplicación de forma horizontal, equilibrando el tráfico automáticamente.

Despliegue y Configuración

Aplicación Web Mi Tiendita (Inventario de Productos) básico

Stack Tecnológico: (ReactJs + FastAPI + MariaDB + DynamoDB + AWS S3).

Inventario Agregar Producto Realizar Backup

Buscar por nombre...

Total de productos 🛒
3

Costo total de productos \$
\$103572

Nombre del Producto	Precio	Stock	Descripción	Acciones
7Up	9	8	bebida refrescante	Editar Eliminar
Seven UP	4000	4	Otra bebida refrescante	Editar Eliminar
Colombiana	3500	25	Colombiana la Nuestra	Editar Eliminar

Instancias BackEnd (3)

```
docker-compose.yml 1.43 KiB
1 version: "3.8"
2
3 services:
4   fastapi_app:
5     image: juadadev/backendproducts:latest # Imagen desde Docker Hub
6     deploy:
7       replicas: 3
8       restart_policy:
9         condition: on-failure
10    secrets:
11      - DB_HOST
```

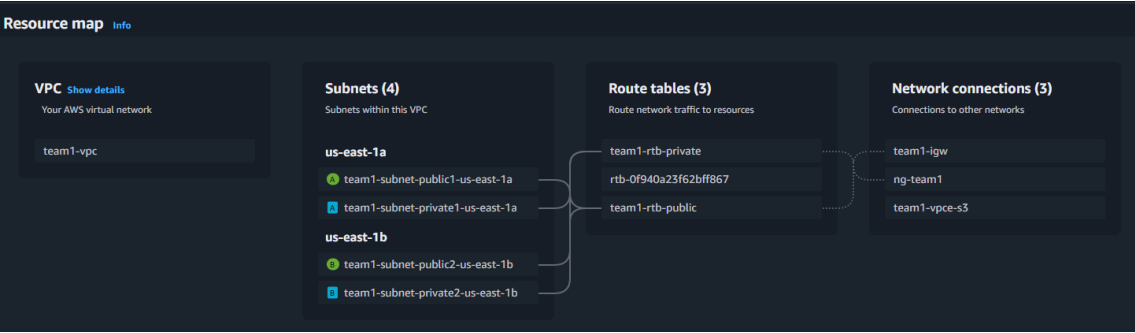
Instancias FrontEnd (2)

```
docker-compose.yml 323 B
1 version: "3.8"
2
3 services:
4   frontend_app:
5     image: juadadev/frontendproducts:latest
6     deploy:
7       replicas: 2
8       restart_policy:
9         condition: on-failure
10    ports:
11      - "80:80" # Puedes cambiar a otro puerto si ya está usado
12    networks:
13      - app_network
14
15 networks:
16   app_network:
17     external: true
18
```

Recursos de AWS

VPC: Mapa de Recursos

team1-vpc: [vpc-0f8a2bbb79fe68055](#)



Se observan los recursos ligados a la VPC (subnets, NAT Gateway, tabla de rutas, Internet Gateway, etc)

Security Group: team1-sg

sg-07e404c330a7fc61c - team1-sg

Actions

Details

Security group name

team1-sg

Security group ID

sg-07e404c330a7fc61c

Description

Grupo para acceder a los servicios

VPC ID

vpc-0f8a2bbb79fe68055

Owner

586794478167

Inbound rules count

9 Permission entries

Outbound rules count

2 Permission entries

Inbound rules

Outbound rules

Sharing - new

VPC associations - new

Tags

Inbound rules (9)

Manage tags

Edit inbound rules

Search

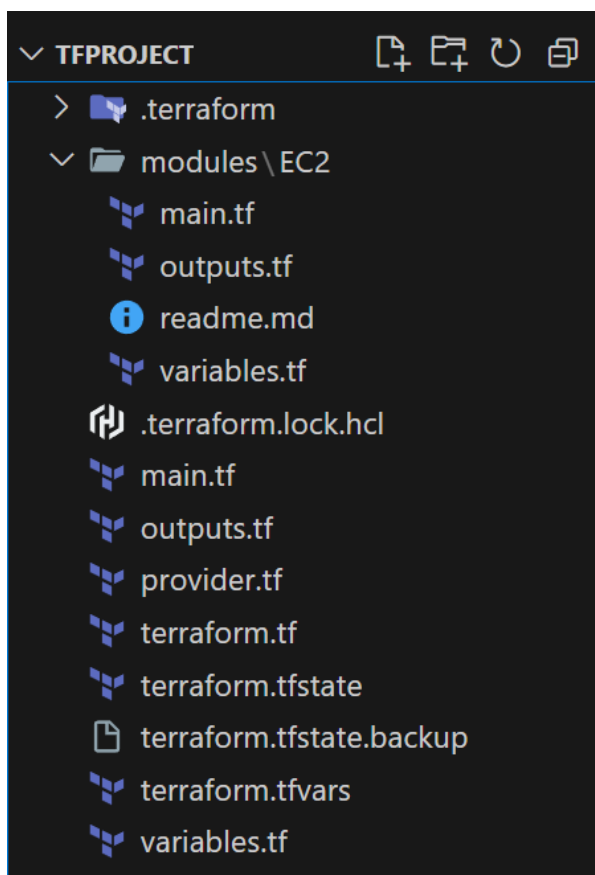
name	Security group rule ID	IP version	Type	Protocol	Port range	Source	Description
	sg-0d441f0439436789c	-	Custom TCP	TCP	8000	sg-07e404c330a7fc61c ...	-
	sg-06fc263bfc4a501dc	IPv4	SSH	TCP	22	0.0.0.0/0	-
	sg-073035bd35d044f05	-	Custom TCP	TCP	7946	sg-07e404c330a7fc61c ...	Descubrimiento de nodos
	sg-0c4ba2fe75b82cd09	IPv4	HTTP	TCP	80	0.0.0.0/0	HTTP acceso a la app
	sg-0cea9d062c467c3c6	IPv4	HTTPS	TCP	443	0.0.0.0/0	HTTP acceso a la app
	sg-03cec0aefcc601b49	-	Custom TCP	TCP	2377	sg-07e404c330a7fc61c ...	Gestion del cluster Swa...
	sg-07bed2e7c387ef003	-	Custom UDP	UDP	7946	sg-07e404c330a7fc61c ...	Descubrimiento de nodos
	sg-0b78c217a40fe3253	-	Custom UDP	UDP	4789	sg-07e404c330a7fc61c ...	Trafico de red overlay
	sg-0338a9bab64b687a5	IPv4	MYSQL/Aurora	TCP	3306	0.0.0.0/0	-

Inbound rules o reglas de entrada para el grupo de seguridad, las especificadas en el documento y adicionales para la correcta ejecución del aplicativo.

Instancias EC2

Name	Instance ID	Instance state	Instance type	Status check
EC2-Instance Worker 1	i-0368502c1cdf99174	Running	t2.micro	2/2 checks passed
Bastion Host	i-06933e162d3df4153	Running	t2.micro	2/2 checks passed
EC2-Instance Manager	i-07b2fd526d02609ed	Running	t2.micro	2/2 checks passed
EC2-Instance Worker 2	i-0cd9cc17903d9ce45	Running	t2.micro	2/2 checks passed

Creadas a partir de IaC con un proyecto de Terraform estructurado de la siguiente manera:



Definiendo instancias AMI Linux t2.micro con 8 gb de almacenamiento y ligadas a la VPC y subredes privadas (Manager y Worker 1 ---> subnet-private1) y Worker 2 ---> subnet-private

ALB: Application Load Balancer

ALB-team1

▼ Details

Load balancer type

Application

Scheme

Internet-facing

Status

Active

Hosted zone

Z35SXDOTRQ7X7K

VPC

vpc-0f8a2bbb79fe68055

Availability Zones

subnet-0a83910101026db68 us-east-1b (use1-az2)

subnet-01327099393306e4f us-east-1a (use1-az1)

Load balancer IP address type

IPv4

Date created

April 8, 2025, 00:40 (UTC-05:00)

Load balancer ARN

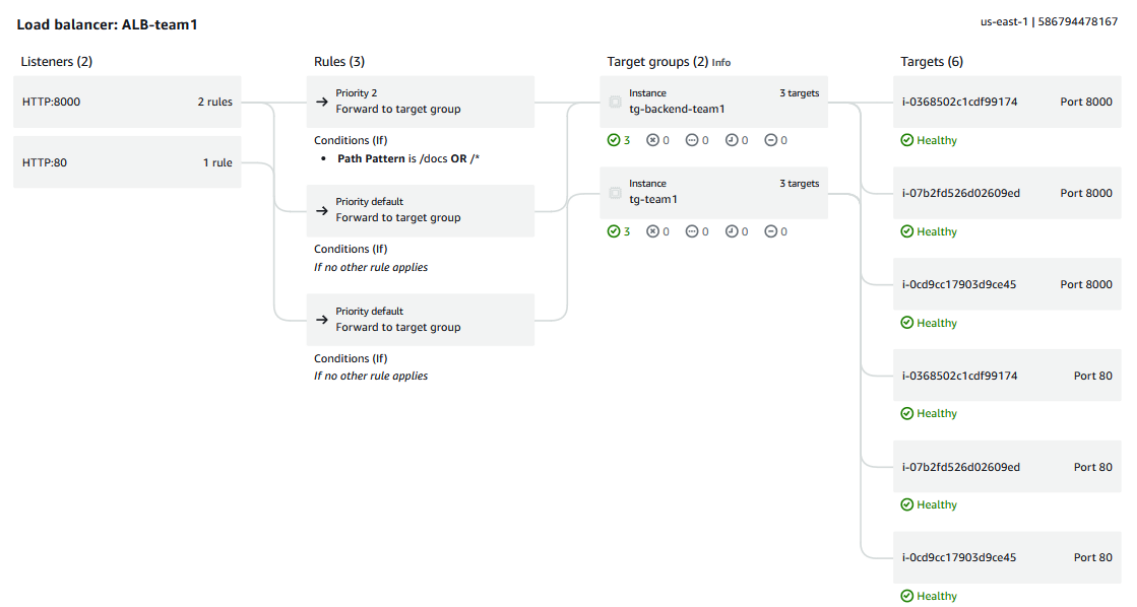
arn:aws:elasticloadbalancing:us-east-1:586794478167:loadbalancer/app/ALB-team1/12697e9dd5b841b6

DNS name Info

ALB-team1-1512890093.us-east-1.elb.amazonaws.com (A Record)

Actions

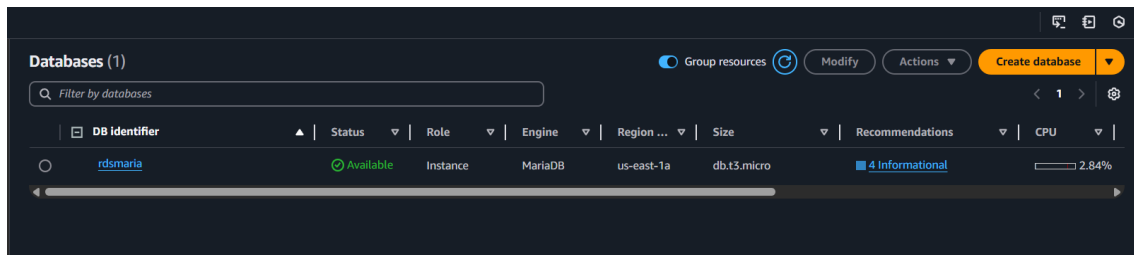
Para distribuir la carga del aplicativo entre las 3 instancias de EC2



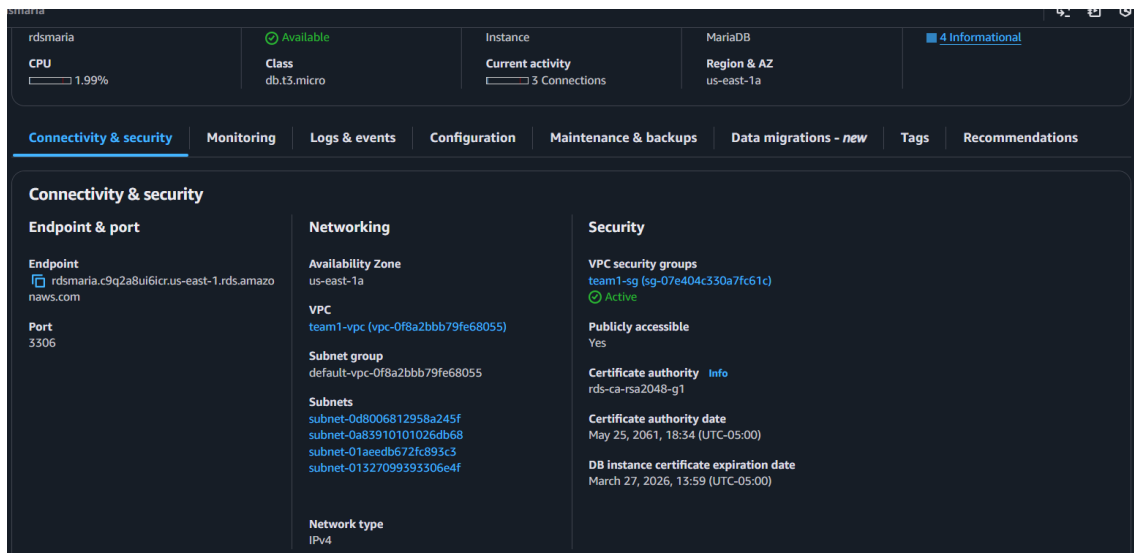
Mapa de Recursos ALB, junto con Target Groups refiriendo a las instancias, Listeners y Rules.

Base de Datos Amazon RDS

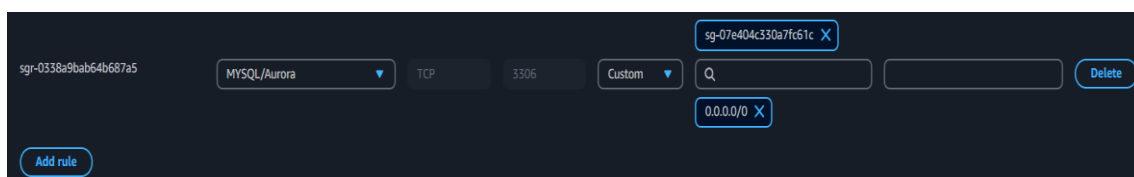
Para el desarrollo del ejercicio, se optó por trabajar con una instancia DB del motor de base de datos relacional MariaDB, debido a la alta comprensión y experiencia de los integrantes en instancias de Tipo MySQL y relacionadas.



A continuación, un detalle más a fondo de las propiedades de la instancia:

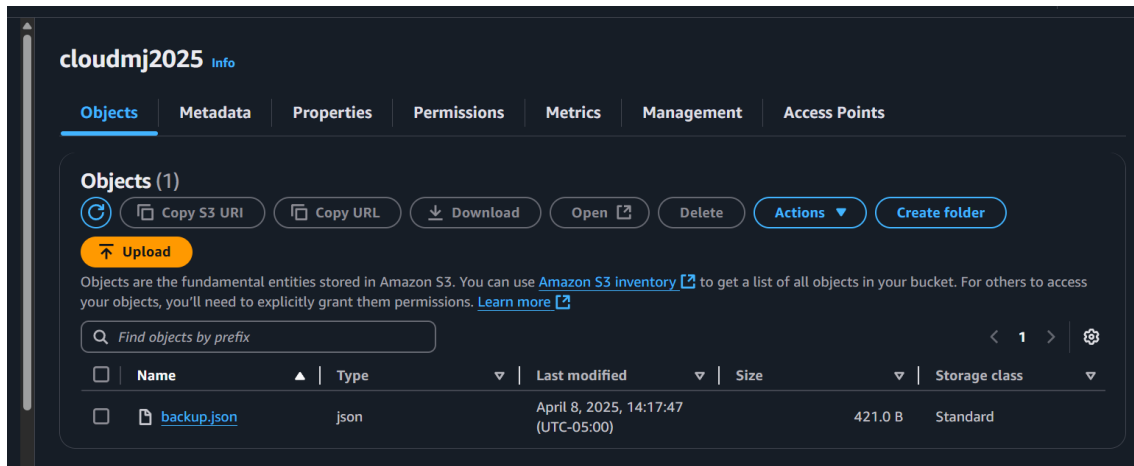


Donde se observa la VPC a la cual está relacionada y el grupo de seguridad, cabe recalcar que debe haber una regla para permitir el tráfico entrante desde el puerto 3306 hacia nuestra instancia:



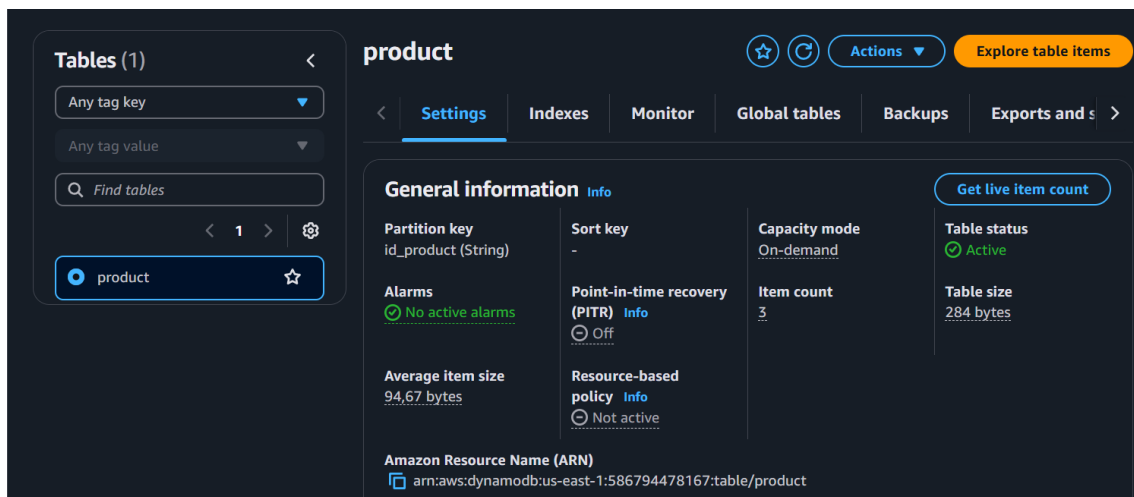
Permite la conexión desde cualquier ip a la instancia de RD

S3 Bucket: cloudmj2025



Bucket de S3 para almacenar la copia de seguridad de las bases de datos.

Instancia de DynamoDB



Para almacenamiento escalable en la nube de manera NOSQL

Capturas de Pantalla del Clúster y Pruebas Realizadas

Creación del Clúster con Docker Swarm

Comando “Docker swarm init” para establecer el nodo (instancia EC2) como Manager y obtener el token a usar en los nodos Worker

```

[ec2-user@ip-10-40-61-76 ~]$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
[ec2-user@ip-10-40-61-76 ~]$ sudo docker swarm init
Swarm initialized: current node (vnhtlh1734whedwdeiawvqh63) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4otm9b5977cf4nw5by5caw3u8a2lwd1g1q2ze39qoj81v3v9co-59git1bm7wq1xpg2jzs9f h6ds 10.40.61.76:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

[ec2-user@ip-10-40-61-76 ~]$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
[ec2-user@ip-10-40-61-76 ~]$ sudo docker node ls
ID                                HOSTNAME
vnhtlh1734whedwdeiawvqh63 *      ip-10-40-61-76.ec2.internal      Ready   Active   Leader
xxpsak5a7g6w0jnpk40o0hygc         ip-10-40-61-243.ec2.internal     Ready   Active
[ec2-user@ip-10-40-61-76 ~]$ ls
[ec2-user@ip-10-40-61-76 ~]$ sudo docker node ls
ID                                HOSTNAME
vnhtlh1734whedwdeiawvqh63 *      ip-10-40-61-76.ec2.internal      Ready   Active   Leader
xxpsak5a7g6w0jnpk40o0hygc         ip-10-40-61-243.ec2.internal     Ready   Active
nyafzka4udmmczcx0q3t6he1n         ip-10-40-62-146.ec2.internal     Ready   Active
[ec2-user@ip-10-40-61-76 ~]$

```

Deploy de contenedor del Backend y correcta ejecución desde el nodo Manager

Comando “docker stack deploy –c docker-compose-backend.yml backend_stack”

para desplegar el servicio a través de Docker Swarm, se ejecutan los mismos pasos

para el despliegue del contenedor del servicio de FrontEnd

```

nvim nvim ec2-user@ip-10- ec2-user@ip-10- ec2-user@ip-10- nvim token.txt nvim swarm + Update
~/Desktop
ssh -i "clave-team1.pem" ec2-user@ec2-18-215-188-74.compute-1.amazonaws.com
[ec2-user@ip-10-40-61-76 ~]$ sudo docker stack deploy -c docker-compose-backend.yml backend_stack
Creating service backend_stack_fastapi_app
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service ls
ID                                NAME                                MODE                                REPLICAS    IMAGE                                PORTS
jsb3hguydydnk backend_stack_fastapi_app replicated 3/3          juadadev/backendproducts:latest    *:8000->8000/tcp
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service logs backend_stack_fastapi_app
backend_stack_fastapi_app.2.she8xqxpqb3@ip-10-40-61-76.ec2.internal | INFO: Started server process [1]
backend_stack_fastapi_app.2.she8xqxpqb3@ip-10-40-61-76.ec2.internal | INFO: Waiting for application startup.
backend_stack_fastapi_app.2.she8xqxpqb3@ip-10-40-61-76.ec2.internal | INFO: Application startup complete.
backend_stack_fastapi_app.2.she8xqxpqb3@ip-10-40-61-76.ec2.internal | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
backend_stack_fastapi_app.1.gt4zyllilyvd@ip-10-40-61-243.ec2.internal | INFO: Started server process [1]
backend_stack_fastapi_app.1.gt4zyllilyvd@ip-10-40-61-243.ec2.internal | INFO: Waiting for application startup.
backend_stack_fastapi_app.1.gt4zyllilyvd@ip-10-40-61-243.ec2.internal | INFO: Application startup complete.
backend_stack_fastapi_app.1.gt4zyllilyvd@ip-10-40-61-243.ec2.internal | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
backend_stack_fastapi_app.3.po8n9vhssoeb@ip-10-40-61-243.ec2.internal | INFO: Started server process [1]
backend_stack_fastapi_app.3.po8n9vhssoeb@ip-10-40-61-243.ec2.internal | INFO: Waiting for application startup.
backend_stack_fastapi_app.3.po8n9vhssoeb@ip-10-40-61-243.ec2.internal | INFO: Application startup complete.
backend_stack_fastapi_app.3.po8n9vhssoeb@ip-10-40-61-243.ec2.internal | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
[ec2-user@ip-10-40-61-76 ~]$ curl http://localhost:8000/docs
^C
[ec2-user@ip-10-40-61-76 ~]$

```

Listado de Servicios y Réplicas con docker service ls

```

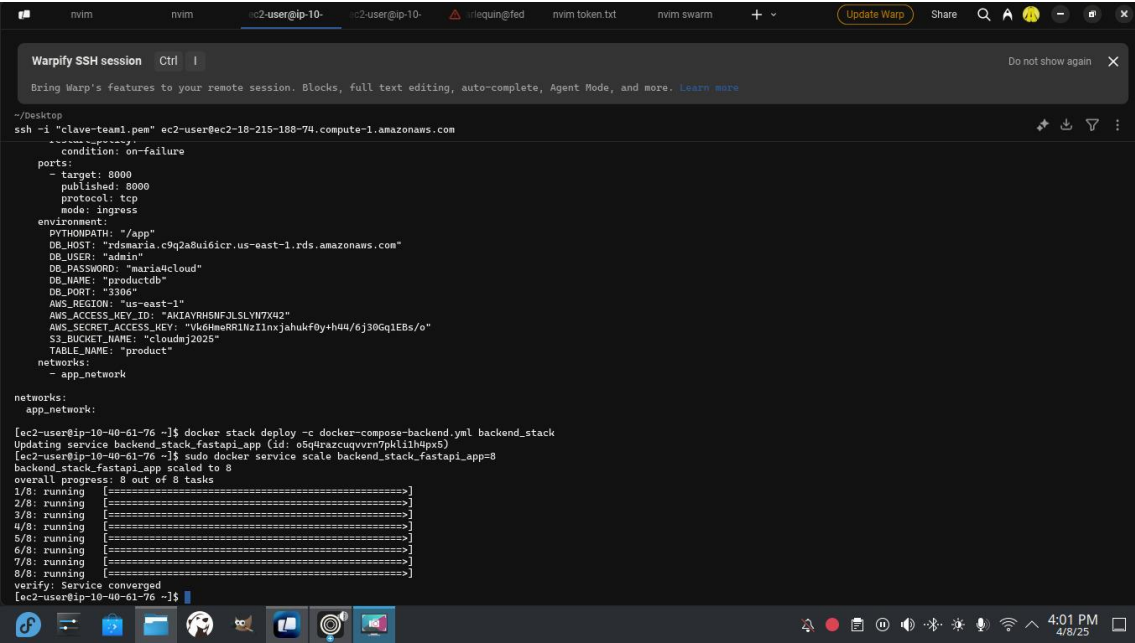
nvim nvim ec2-user@ip-10- ec2-user@ip-10- mequin@fed nvim token.txt nvim swarm + Update
[ec2-user@ip-10-40-61-76 ~]$ sudo docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
juadadev/frontendproducts <none>       06d8770770a      25 minutes ago  48.1MB
juadadev/backendproducts <none>       b34597b5c258     About an hour ago 218MB
[ec2-user@ip-10-40-61-76 ~]$ sudo docker rmi -f 06d
sudo: rmi: command not found
[ec2-user@ip-10-40-61-76 ~]$ sudo docker rmi -f 06d
Error response from daemon: conflict: unable to delete 06d8770770a (cannot be forced) - image is being used by running container ae6b147939f1
[ec2-user@ip-10-40-61-76 ~]$ sudo docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
juadadev/frontendproducts <none>       06d8770770a      25 minutes ago  48.1MB
juadadev/backendproducts <none>       b34597b5c258     About an hour ago 218MB
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service ls
ID                                NAME                                MODE                                REPLICAS    IMAGE                                PORTS
o5q4razcuqv backend_stack_fastapi_app replicated 3/3          juadadev/backendproducts:latest
zb9zqpm40a frontend_stack_frontend_app replicated 2/2          juadadev/frontendproducts:latest
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service rm zb
zb
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service ls
ID                                NAME                                MODE                                REPLICAS    IMAGE                                PORTS
o5q4razcuqv backend_stack_fastapi_app replicated 3/3          juadadev/backendproducts:latest
[ec2-user@ip-10-40-61-76 ~]$

```

Se observan 3 réplicas para el servicio de BackEnd y 2 réplicas para el servicio de FrontEnd.

Pruebas de Escalabilidad

“docker service scale” para aumentar el número de réplicas del servicio BackEnd



The screenshot shows a terminal window with a Warpify SSH session. The user is logged into an EC2 instance. They run the command `docker stack deploy -c docker-compose-backend.yml backend_stack` to update the service. Then, they run `sudo docker service scale backend_stack_fastapi_app=8` to scale the service to 8 replicas. The output shows the progress of scaling, with 8 tasks running and the service converging.

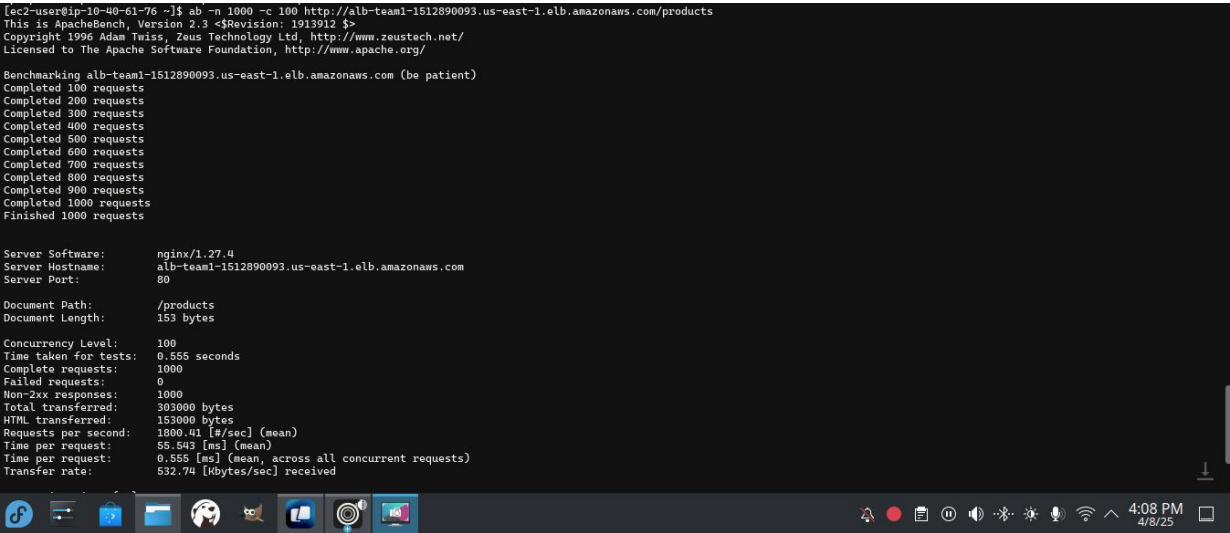
```
~/desktop
ssh -i "clave-team1.pem" ec2-user@ec2-18-215-188-74.compute-1.amazonaws.com

condition: on-failure
ports:
  - target: 8000
    published: 8000
    protocol: tcp
    mode: ingress
environment:
  PYTHONPATH: "/app"
  DB_HOST: "rdsmaria.c9q2a8ui6icr.us-east-1.rds.amazonaws.com"
  DB_USER: "admin"
  DB_PASSWORD: "mariaHeloud"
  DB_NAME: "productdb"
  DB_PORT: "3306"
  AWS_REGION: "us-east-1"
  AWS_ACCESS_KEY_ID: "AKIAVHSMFJLSLVW7X42"
  AWS_SECRET_ACCESS_KEY: "V66Hm0RR1NzIlnxjahukf0y+u4/6j30Gq1EBs/o"
  S3_BUCKET_NAME: "cloudmj2025"
  TABLE_NAME: "product"
networks:
  - app_network

networks:
  app_network:

[ec2-user@ip-10-40-61-76 ~]$ docker stack deploy -c docker-compose-backend.yml backend_stack
Updating service backend_stack_fastapi_app (id: 05q4razcuqvzv7pRilhapx3)
[ec2-user@ip-10-40-61-76 ~]$ sudo docker service scale backend_stack_fastapi_app=8
backend_stack_fastapi_app scaled to 8
overall progress: 8 out of 8 tasks
1/8: running [=====>]
2/8: running [=====>]
3/8: running [=====>]
4/8: running [=====>]
5/8: running [=====>]
6/8: running [=====>]
7/8: running [=====>]
8/8: running [=====>]
verify: Service converged
[ec2-user@ip-10-40-61-76 ~]$
```

Pruebas con Apache Benchmark



The screenshot shows a terminal window where the user runs the Apache Benchmark (ab) command: `ab -n 1000 -c 100 http://alb-team1-1512890093.us-east-1.elb.amazonaws.com/products`. The output displays the progress of the benchmark, showing 1000 requests completed. It then provides detailed statistics about the benchmark, including server software, hostname, port, document path, concurrency level, time taken, requests per second, and transfer rate.

```
[ec2-user@ip-10-40-61-76 ~]$ ab -n 1000 -c 100 http://alb-team1-1512890093.us-east-1.elb.amazonaws.com/products
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking alb-team1-1512890093.us-east-1.elb.amazonaws.com (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests

Server Software:      nginx/1.27.4
Server Hostname:      alb-team1-1512890093.us-east-1.elb.amazonaws.com
Server Port:          80

Document Path:        /products
Document Length:       153 bytes

Concurrency Level:     100
Time taken for tests:   0.555 seconds
Complete requests:     1000
Failed requests:        0
Non-2xx responses:     1000
Total transferred:     303000 bytes
HTML transferred:      153000 bytes
Requests per second:   1800.41 [#/sec] (mean)
Time per request:      55.543 [ms] (mean)
Time per request:      0.555 [ms] (mean, across all concurrent requests)
Transfer rate:          532.74 [Kbytes/sec] received
```

Donde se simulan # de peticiones (1000) y # de peticiones concurrentes (100) al endpoint del servicio de Backend, resultando en métricas que podemos analizar de la siguiente manera:

```
Connection Times (ms)
      min  mean[+/-sd] median  max
Connect:    1    17   4.4    18    22
Processing:  8    36  15.7    40   103
Waiting:    5    34  15.9    39   102
Total:      9    53  15.6    58   122

Percentage of the requests served within a certain time (ms)
 50%    58
 66%    60
 75%    61
 80%    61
 90%    63
 95%    81
 98%    97
 99%   102
100%   122 (longest request)
[ec2-user@ip-10-40-61-76 ~]$ i
```

Métricas Resultantes

✅ 1. Rendimiento general (¡muy bueno!)		
Métrica	Valor	Comentario
📄 Total de peticiones	1000	Buen volumen para una prueba inicial
👤 Concurrencia	100	Simula 100 usuarios accediendo al mismo tiempo
🕒 Tiempo total	0.555 segundos	Muy bajo para 1000 requests
⚡ Requests por segundo	1800.41 req/sec	Excelente capacidad de respuesta del sistema
🧠 Tiempo medio por request	55 ms	Rápido, ideal para experiencia de usuario fluida
📦 Transferencia	532.74 KB/s	Normal, pero depende del tamaño de respuesta
❌ Failed Requests	0	✅ Todo bien a nivel de red/conexión

Donde se obtienen métricas de resultados bastante satisfactorias que demuestran un buen soporte a la escalabilidad del aplicativo.

Códigos Docker

Backend

- Dockerfile

```
1 FROM python:3.12-slim
2
3 WORKDIR /app
4
5 ENV PYTHONPATH=/app
6
7 COPY requirements.txt .
8
9 RUN pip install --no-cache-dir --upgrade -r requirements.txt
10
11 COPY . .
12
13 EXPOSE 8000
14
15 CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
16
```

- Docker-compose

```
1 version: "3.8"
2
3 services:
4   fastapi-app:
5     image: juadadev/backendproducts:latest # Imagen desde Docker Hub
6     deploy:
7       replicas: 3
8       restart_policy:
9         condition: on-failure
10     secrets:
11       - DB_HOST
12       - DB_USER
13       - DB_PASSWORD
14       - DB_NAME
15       - DB_PORT
16       - AWS_REGION
17       - AWS_ACCESS_KEY_ID
18       - AWS_SECRET_ACCESS_KEY
19       - S3_BUCKET_NAME
20       - TABLE_NAME
21     ports:
22       - target: 8000
23         published: 8000
24         protocol: tcp
25     mode: host
26     environment:
27       PYTHONPATH: "/app"
28       DB_HOST_FILE: "/run/secrets/DB_HOST"
29       DB_USER_FILE: "/run/secrets/DB_USER"
30       DB_PASSWORD_FILE: "/run/secrets/DB_PASSWORD"
31       DB_NAME_FILE: "/run/secrets/DB_NAME"
32       DB_PORT_FILE: "/run/secrets/DB_PORT"
33       AWS_REGION_FILE: "/run/secrets/AWS_REGION"
34       AWS_ACCESS_KEY_ID_FILE: "/run/secrets/AWS_ACCESS_KEY_ID"
35       AWS_SECRET_ACCESS_KEY_FILE: "/run/secrets/AWS_SECRET_ACCESS_KEY"
36       S3_BUCKET_NAME_FILE: "/run/secrets/S3_BUCKET_NAME"
37       TABLE_NAME_FILE: "/run/secrets/TABLE_NAME"
38     networks:
39       - app_network
40
41 networks:
42   app_network:
43
44 secrets:
45   DB_HOST:
46     external: true
47   DB_USER:
48     external: true
49   DB_PASSWORD:
50     external: true
51   DB_NAME:
52     external: true
53   DB_PORT:
54     external: true
55   AWS_REGION:
56     external: true
57   AWS_ACCESS_KEY_ID:
58     external: true
59   AWS_SECRET_ACCESS_KEY:
60     external: true
61   S3_BUCKET_NAME:
62     external: true
63   TABLE_NAME:
64     external: true
65
```

Frontend

- Docker-compose:

```
🐳 docker-compose.yml 📄 323 B

1  version: "3.8"
2
3  services:
4    frontend_app:
5      image: juadadev/frontendproducts:latest
6      deploy:
7        replicas: 2
8        restart_policy:
9          condition: on-failure
10     ports:
11       - "80:80" # Puedes cambiar a otro puerto si ya está usado
12     networks:
13       - app_network
14
15   networks:
16     app_network:
17       external: true
18
```

- Dockerfile

```
1  # Etapa 1: Build del frontend
2  FROM node:22.14.0-slim AS builder
3
4  # Establecer el directorio de trabajo
5  WORKDIR /app
6
7  # Definir las variables de entorno
8  ARG VITE_BACKEND_URL
9  ENV VITE_BACKEND_URL=$VITE_BACKEND_URL
10
11 # Copiar dependencias y archivos necesarios
12 COPY package*.json ./
13 COPY postcss.config.js tailwind.config.js vite.config.js ./
14 COPY src ./src
15 COPY index.html ./
16
17 # Instalar dependencias y construir
18 RUN npm install
19 RUN npm run build
20
21 # Etapa 2: Servidor estático con NGINX
22 FROM nginx:alpine
23
24
25 # Copiar el build al directorio por defecto de NGINX
26 COPY --from=builder /app/dist /usr/share/nginx/html
27
28 # Copiar configuración personalizada (opcional)
29 # COPY nginx.conf /etc/nginx/nginx.conf
30
31 # Exponer el puerto 80
32 EXPOSE 80
33
34 # Iniciar NGINX
35 CMD ["nginx", "-g", "daemon off;"]
36
```

Enlace Repositorios

Frontend:

<https://gitlab.com/unicesarcol/cloud/cloud-scadena-202501-mj/team1/frontendproduct2>

Backend:

<https://gitlab.com/unicesarcol/cloud/cloud-scadena-202501-mj/team1/backendproducts>

Terraform Project para IaC

<https://gitlab.com/unicesarcol/cloud/cloud-scadena-202501-mj/team1/terraformproducts.git>

Conclusión

Con esta actividad se aplicaron conocimiento de infraestructura en **AWS**, contenedores en Docker Swarm, redes, bases de datos y almacenamiento en la nube desarrollando habilidades en un entorno real.