Universidad de Las Palmas de Gran Canaria

Escuela de Ingeniería Informática

Práctica Entregable

Bitcoin Positions Tracker

Diseño de Aplicaciones en la Nube

Asignatura: Computación en la Nube

Autor: Sergio Acosta Quintana

5 de noviembre de 2025

Índice

1. Introducción

El proyecto implementa un sistema de gestión de posiciones de trading de Bitcoin mediante dos arquitecturas AWS:

- **ECS Fargate (no desacoplada):** Contenedor Docker con Express.js
- **Lambda (desacoplada):** 5 funciones serverless independientes

Ambas cumplen los requisitos CRUD, utilizan DynamoDB, exponen API REST documentada con Swagger, y se despliegan mediante CloudFormation.

Repositorio: https://github.com/SergioAcostaTer/cloud-computing-1

2. Esquemas de Arquitectura

2.1. Diagrama Comparativo

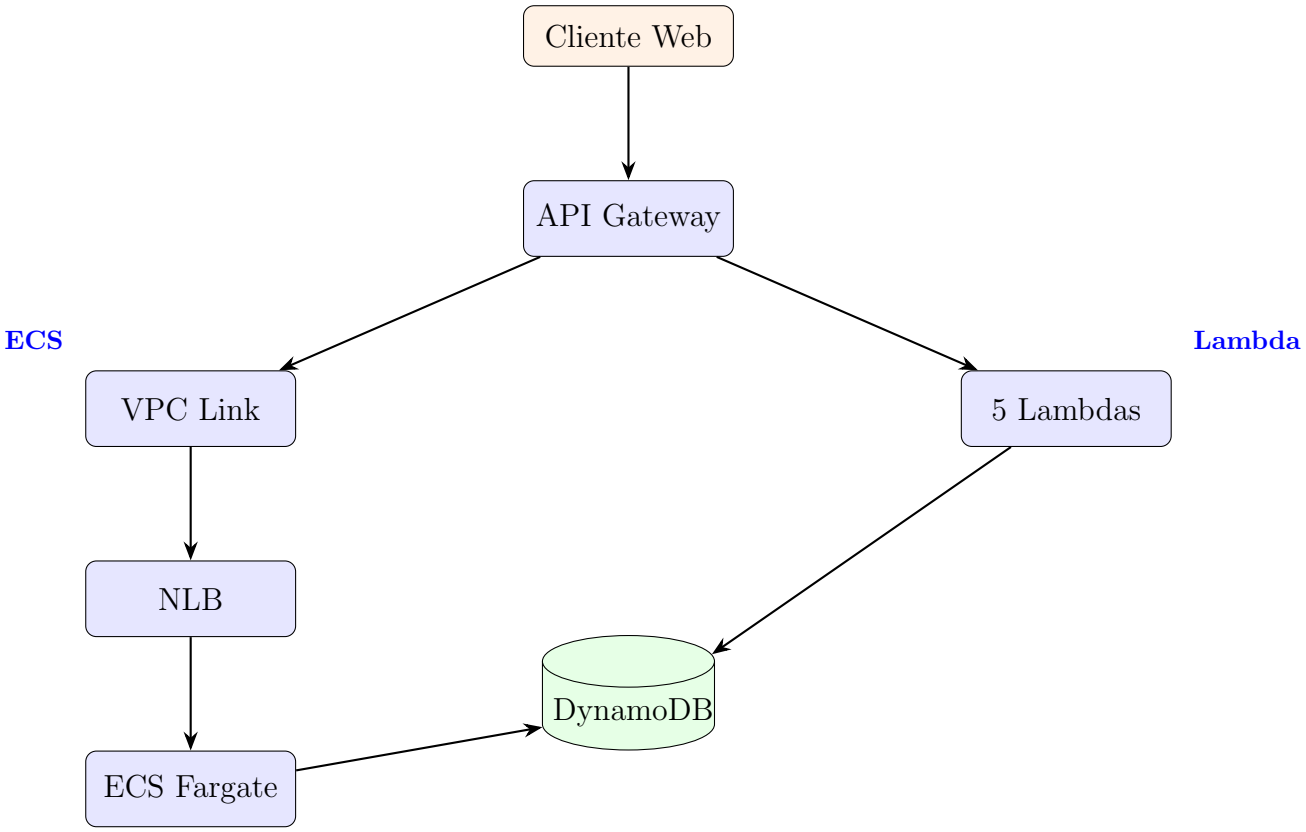

Figura 1: Arquitecturas implementadas (ECS vs Lambda)

2.2. Descripción de Componentes

2.2.1. Versión ECS (No Desacoplada)

- **API Gateway:** Punto de entrada con API Key

- **VPC Link:** Conexión segura entre API Gateway y NLB privado
- **Network Load Balancer:** Distribución de tráfico interno
- **ECS Fargate:** Contenedor Docker (Node.js + Express)
- **DynamoDB:** Base de datos NoSQL (tabla BitcoinPositions)

2.2.2. Versión Lambda (Desacoplada)

- **API Gateway:** REST API con integración AWS_PROXY
- **Lambda Functions (5):**
  - `btc-crud-operations` - Create, Update, Delete
  - `btc-read-operations` - Get all, Get by ID
  - `btc-openapi-docs` - Documentación Swagger
  - `btc-health-check` - Health endpoint
  - `btc-root-handler` - Root endpoint
- **DynamoDB:** Diferente nombre de tabla pero misma definición

3. Cumplimiento de Requisitos

3.1. Base de Datos (1.1)

DynamoDB tabla "BitcoinPositions":

- Atributos: `id, symbol, quantity, type, entry, date` (6 atributos)
- Clave primaria: `id` (String)
- Billing: PAY_PER_REQUEST (escalado automático)

3.2. Computación y Despliegue (1.2)

Versión ECS:

- Contenedor Docker con Node.js 18
- Deployment automatizado con Makefile: `make build && make push`
- CloudFormation despliega ECS cluster, task definition y service

Versión Lambda:

- 5 funciones desplegadas desde S3 (`lambda-code.zip`)
- Script PowerShell automatiza empaquetado y deploy

3.3. Desacoplamiento (1.3)

Lambda cumple desacoplamiento:

- Funciones independientes comunicándose via DynamoDB
- Cada función tiene responsabilidad única
- API Gateway orquesta invocaciones

3.4. Operaciones CRUD

Cuadro 1: Endpoints implementados

Método	**Endpoint**	**Operación**
POST	/positions	Create (nuevo elemento)
GET	/positions	Read All (todos los elementos)
GET	/positions/{id}	Read (elemento por ID)
PUT	/positions/{id}	Update (actualizar existente)
DELETE	/positions/{id}	Delete (eliminar elemento)

3.5. Puntos Adicionales Conseguidos

- **+3 puntos:** YAML despliega código automáticamente
  - ECS: Imagen Docker desde ECR
  - Lambda: Código desde S3 bucket
- **+1 punto:** Documentación automática en `/openapi.json`
- **+1 punto:** Frontend completo en S3 con CRUD funcional

Total: 10/10 puntos

4. Análisis de Costes

4.1. Supuestos

- Tráfico: 1 millón de peticiones/mes
- Región: us-east-1 (N. Virginia)
- Lambda: 200 ms promedio, 256 MB RAM
- ECS: 1 tarea (0.25 vCPU, 0.5 GB) ejecutándose 24/7

Cuadro 2: Comparativa de costes sin Free Tier (noviembre 2025, us-east-1)

Servicio	**ECS**		**Lambda**	
	Mes	**Año**	**Mes**	**Año**
Computación	\$9.00	\$108.00	\$0.83	\$9.96
API Gateway	\$3.50	\$42.00	\$3.50	\$42.00
NLB / VPC Link	\$38.70	\$464.40	-	-
DynamoDB	\$1.00	\$12.00	\$1.00	\$12.00
Transfer + Logs	\$5.00	\$60.00	\$2.00	\$24.00
TOTAL	**\$57.20**	**\$686.40**	**\$7.53**	**\$90.36**
Ahorro	-	-	**87 %**	**87 %**

4.2. Costes Mensuales y Anuales

Supuestos de cálculo

Los valores de la tabla anterior se basan en tarifas reales de **AWS (noviembre 2025, región us-east-1)**, sin aplicar capa gratuita (*Free Tier*). Los siguientes supuestos y volúmenes de uso se emplearon para la estimación de costes mensuales y anuales:

- **Tráfico:** 1 000 000 peticiones REST al mes.
- **Región:** `us-east-1 (N. Virginia)`.
- **Arquitectura ECS Fargate:**
  - 1 tarea Fargate ejecutándose 24/7 con **0.25 vCPU** y **0.5 GB RAM**.
  - **Network Load Balancer (NLB)** activo todo el mes (1 LCU).
  - **VPC Link** asociado a API Gateway (activo 730 h/mes).
  - API Gateway REST con 1 millón de solicitudes mensuales.
  - DynamoDB con 1 millón de lecturas y 0.2 millones de escrituras mensuales.
  - Logs y transferencia moderada (10 GB/mes).
- **Arquitectura Lambda desacoplada:**
  - 5 funciones Lambda independientes.
  - Cada función: **256 MB de memoria**, **200 ms** por invocación.
  - 1 millón de invocaciones totales al mes (repartidas entre funciones).
  - API Gateway REST con 1 millón de peticiones mensuales.
  - DynamoDB con el mismo volumen de lecturas/escrituras que ECS.
  - Logs y transferencia estimada 5 GB/mes.
- **Fuentes de precios AWS (noviembre 2025):**
  - Fargate: \$0.04048/vCPU·h, \$0.004445/GB·h
  - Lambda: \$0.00001667/GB·s y \$0.20/millón de invocaciones

- API Gateway REST: \$3.50/millón de peticiones
- NLB: \$0.0225/hora + \$0.008/LCU·h
- VPC Link: \$0.0225/hora
- DynamoDB (on-demand): \$0.25/millón lecturas, \$1.25/millón escrituras

Estos valores representan un escenario empresarial típico de carga media y sirven para comparar el **coste base de operación mensual** entre ambas arquitecturas.

4.3. Justificación

ECS: Presenta costes fijos elevados (Fargate, NLB y VPC Link) que no dependen del tráfico. Incluso con baja demanda, el coste base ronda los \$57/mes.

Lambda: Ofrece un modelo de pago por uso real. Para 1 M de peticiones mensuales, el coste total se mantiene por debajo de \$8/mes, y escala de forma lineal con la carga. El punto de equilibrio frente a ECS se alcanza aproximadamente a partir de 100 M de invocaciones mensuales.

Frontend S3: Coste aproximado de \$0.12/mes, despreciable y no incluido en las tablas.

5. Ficheros CloudFormation

5.1. backend/ecs/deploy.yml

550+ líneas definiendo:

- DynamoDB Table
- VPC, Security Groups, Subnets (parámetros)
- Network Load Balancer + Target Group
- ECS Cluster, Task Definition, Service
- API Gateway REST con VPC Link
- Endpoints (GET, POST, PUT, DELETE) + CORS
- API Key + Usage Plan

Parámetros: `ECRImage`, `VpcId`, `SubnetIds`

5.2. backend/lambda/deploy.yml

550+ líneas definiendo:

- DynamoDB Table (BitcoinPositionsDecoupled)
- 5 Lambda Functions + Permissions
- API Gateway REST (AWS_PROXY)
- Endpoints + métodos OPTIONS (CORS)
- API Key + Usage Plan

Parámetros: `LambdaCodeBucket`, `LambdaCodeKey`

6. Código de la Aplicación

6.1. Estructura del Proyecto

```
backend/
 ecs/
    app.js            # Express + Swagger
    Dockerfile        # Node 18 Alpine
    deploy.yml        # CloudFormation
    Makefile          # Automatización
 lambda/
     lambdas/
        crud.js       # Create/Update/Delete
        read.js       # Get operations
        openapi.js    # Swagger spec
        health.js     # Health check
        root.js       # Root handler
     deploy.yml        # CloudFormation
     deploy.ps1        # Deploy script

frontend/
 index.html           # Dashboard UI
 scripts.js           # CRUD + WebSocket Binance
 policy.json          # S3 public policy

tests.http               # Colección pruebas REST
```

6.2. Fragmento Express (ECS)

```
app.post("/positions", async (req, res) => {
    const { symbol, quantity, type, entry, date } = req.body;
    const item = { id: uuidv4(), symbol, quantity, type, entry, date };
    await dynamo.put({ TableName: TABLE_NAME, Item: item }).promise();
    res.status(201).json(item);
});
```

6.3. Fragmento Lambda

```
exports.handler = async (event) => {
    if (event.httpMethod === "POST") {
        const body = JSON.parse(event.body);
        const item = { id: uuidv4(), ...body };
        await client.send(new PutItemCommand({
            TableName: TABLE_NAME, Item: marshall(item)
        }));
        return { statusCode: 201, body: JSON.stringify(item) };
    }
};
```

7. Herramientas de Prueba

7.1. tests.http (REST Client)

Colección completa con variables para probar todos los endpoints:

```
### Variables
@host = https://API_ID.execute-api.us-east-1.amazonaws.com/prod
@apiKey = YOUR_API_KEY

### Create Position
POST {{host}}/positions
x-api-key: {{apiKey}}
Content-Type: application/json

{ "symbol": "BTCUSDT", "quantity": 0.5, ... }

### Get All
GET {{host}}/positions
x-api-key: {{apiKey}}
```

7.2. Frontend Web (S3)

Interfaz gráfica completa con:

- Formulario para Create
- Tabla listando posiciones (Read All)
- Botones Edit (Update) y Delete por fila
- Modal para edición individual
- Precio Bitcoin en tiempo real (WebSocket Binance)
- Cálculo automático P&L

8. Conclusiones

8.1. Requisitos Cumplidos

- Base de datos DynamoDB con 6 atributos
- Balanceo (NLB en ECS)
- Escalado automático (Lambda) y manual (ECS)
- Despliegue containerizado (ECS) y serverless (Lambda)
- Desacoplamiento mediante funciones Lambda independientes
- CRUD completo en ambas arquitecturas
- API Gateway obligatorio en ambas

- CloudFormation para infraestructura completa
- Documentación Swagger automática
- Frontend funcional para pruebas

8.2. Lecciones Aprendidas

1. **Lambda es más económico** para tráfico bajo-moderado (76 % ahorro)
2. **VPC Link añade coste fijo** significativo ($22.50/mes) en ECS
3. **Desacoplamiento mejora escalabilidad** pero aumenta complejidad operacional
4. **IaC garantiza reproducibilidad** y facilita troubleshooting

8.3. Recomendación

Para este caso de uso (API CRUD con tráfico moderado), **Lambda es superior** en coste, simplicidad y escalabilidad.

9. Anexos

9.1. Repositorio GitHub

Código completo: https://github.com/SergioAcostaTer/cloud-computing-1
Incluye:

- Backends ECS y Lambda completos
- Templates CloudFormation funcionables
- Frontend con UI moderna
- Tests HTTP automatizados
- Scripts de despliegue (Makefile, PowerShell)
- README con instrucciones detalladas

9.2. Tecnologías AWS Utilizadas

- **Compute:** ECS Fargate, Lambda
- **Database:** DynamoDB
- **Networking:** API Gateway, NLB, VPC Link
- **Storage:** S3, ECR
- **IaC:** CloudFormation
- **Monitoring:** CloudWatch