Práctica Entregable:
Diseño de Aplicaciones en la Nube

Computación en la Nube

Universidad de Las Palmas de Gran Canaria

10 de noviembre de 2025

Índice

1. Introducción

Este documento presenta la memoria técnica de la Práctica Obligatoria 1 de Computación en la Nube, donde se ha desarrollado una aplicación de gestión de personajes (Characters) implementada bajo dos arquitecturas de despliegue diferentes en Amazon Web Services (AWS):

- **Versión Monolítica Acoplada**: Diseñada para desplegarse en AWS ECS Fargate con API Gateway y Network Load Balancer (NLB).
- **Versión Desacoplada Serverless**: Implementada bajo el patrón CRUD Puro con cinco funciones AWS Lambda independientes, con código almacenado en Amazon S3 y expuestas mediante API Gateway.

Ambas arquitecturas comparten la misma base de datos (Amazon DynamoDB) y proporcionan los mismos cinco endpoints CRUD para la gestión de personajes.

2. Arquitecturas de Despliegue

2.1. Versión Monolítica Acoplada (ECS Fargate)

Esta arquitectura sigue el patrón de microservicios sobre contenedores, donde el servicio está siempre activo y disponible. Los componentes principales son:

- **Amazon ECS Fargate**: Plataforma de computación serverless para contenedores que ejecuta las tareas de la aplicación.
- **Network Load Balancer (NLB)**: Balancea el tráfico entre las diferentes tareas ECS, proporcionando alta disponibilidad.
- **API Gateway**: Proporciona una capa de exposición HTTP pública para los endpoints de la aplicación.
- **Amazon ECR**: Almacena las imágenes Docker de la aplicación.
- **Amazon DynamoDB**: Base de datos NoSQL que almacena la información de los personajes.

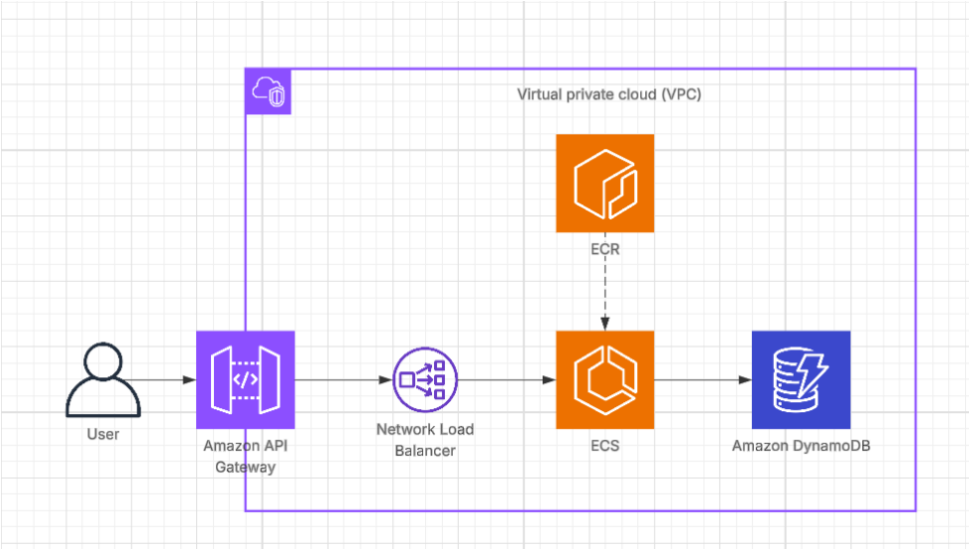

Figura 1: Esquema diagrama versión acoplada

2.2. Versión Desacoplada Serverless (AWS Lambda)

Esta arquitectura es completamente serverless y de pago por uso, descomponiendo la aplicación en cinco funciones independientes (una por operación CRUD):

- **AWS Lambda**: Cinco funciones independientes (CREATE, READ, UPDATE, DELETE, LIST) que implementan la lógica de negocio.
- **Amazon S3**: Almacena el paquete de código de las funciones Lambda.
- **API Gateway**: Expone los endpoints HTTP y se integra directamente con cada función Lambda.
- **Amazon DynamoDB**: Base de datos compartida con la arquitectura acoplada.

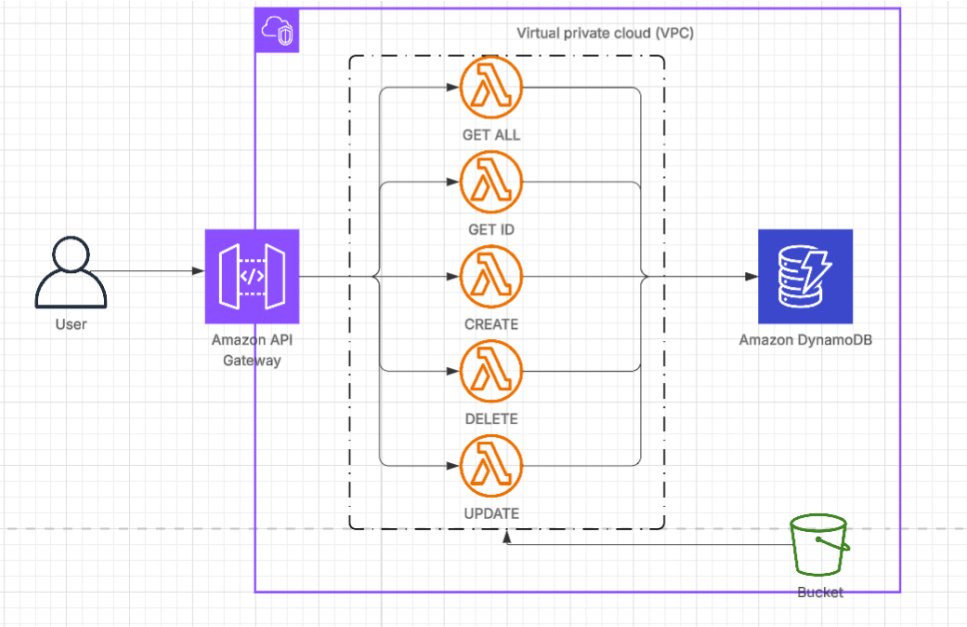

Figura 2: Esquema diagrama versión desacoplada

3. Estructura del Proyecto

La estructura del proyecto está organizada de forma modular para facilitar el mantenimiento y la comprensión de cada arquitectura:

Directorio/Archivo	Contenido Principal	Propósito
`backend/`	Capa de Acceso a Datos	Gestiona la lógica de aplicación y comunicación con DynamoDB
`frontend/`	Código de Interfaz de Usuario	Interfaz HTML básica para interacción con la API
`dynamodb/`	Plantilla CloudFormation	Define la infraestructura de la tabla DynamoDB
`model/`	Definición de Entidades	Define la estructura de datos (objeto Character)
`test/`	Scripts de Pruebas	Verificación funcional de la API
`venv/`	Entorno Virtual Python	Aislamiento de dependencias del proyecto

Cuadro 1: Estructura general del proyecto

3.1. Versión Acoplada - Detalle

Directorio/Archivo	Contenido	Propósito
`acoplada/app/`	Lógica Monolítica	Servidor con 5 endpoints CRUD
`acoplada/config/`	Plantillas CloudFormation	Infraestructura ECS, NLB, API Gateway y ECR
`requirements_acoplada.txt`	Dependencias Python	Librerías necesarias incluyendo framework web
`Dockerfile`	Configuración Docker	Imagen desplegable en ECS Fargate

Cuadro 2: Estructura de la versión acoplada

3.2. Versión Desacoplada - Detalle

Directorio/Archivo	Contenido	Propósito
`desacoplada/app/`	Handlers Lambda	Punto de entrada para cada operación CRUD
`desacoplada/config/`	Plantilla CloudFormation	Infraestructura Serverless completa
`requirements_desacoplada.txt`	Dependencias Python	Librerías esenciales (boto3, pydantic)
`characters_lambda_package.zip`	Paquete de código	Código y dependencias para Lambda

Cuadro 3: Estructura de la versión desacoplada

4. Proceso de Despliegue

4.1. Despliegue de la Versión Acoplada (ECS Fargate)

4.1.1. Prerrequisitos

Antes de iniciar el despliegue, es necesario:

1. Verificar la existencia y corrección de los archivos de configuración: `bd_dynamodb.yml`, `ecr.yml`, `ecs.yml`, `Dockerfile` y `ecs-params.json`.
2. Configurar AWS CLI con las credenciales temporales adecuadas.
3. Asegurar que Docker Desktop esté en ejecución.

Listing 1: Configuración inicial de AWS CLI

```
aws configure
export REGION='{TU_REGION}'
export ACCOUNT_ID='{TU_ID_DE_CUENTA_AWS}'
aws sts get-caller-identity
```

4.1.2. Despliegue de la Base de Datos

El primer paso es crear la tabla DynamoDB mediante CloudFormation:

Listing 2: Creación de la tabla DynamoDB

```
aws cloudformation create-stack \
  --stack-name bdd-stack-p1 \
  --template-body file://acoplada/config/bd_dynamodb.yml \
  --region $REGION

aws cloudformation wait stack-create-complete \
  --stack-name bdd-stack-p1 \
  --region $REGION
```

Una vez creada, obtener el nombre de la tabla para actualizar `ecs-params.json`:

Listing 3: Obtención del nombre de la tabla

```
aws cloudformation describe-stacks \
  --stack-name bdd-stack-p1 \
  --query "Stacks[0].Outputs[?OutputKey=='TableName'].OutputValue" \
  --output text
```

4.1.3. Contenedorización y Registro ECR

Crear el repositorio ECR y construir la imagen Docker:

Listing 4: Creación del repositorio ECR

```
aws cloudformation create-stack \
  --stack-name ecr-stack-p1 \
  --template-body file://acoplada/config/ecr.yml \
  --region $REGION

aws cloudformation wait stack-create-complete \
  --stack-name ecr-stack-p1 \
  --region $REGION
```

Listing 5: Construcción y subida de la imagen Docker

```
export ECR_URI="$ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com/characters-app
    "

aws ecr get-login-password --region $REGION | \
  docker login --username AWS --password-stdin $ECR_URI

docker build -t characters-app .
docker tag characters-app:latest $ECR_URI:latest
docker push $ECR_URI:latest
```

4.1.4. Despliegue de la Infraestructura ECS

Desplegar la pila completa que incluye ECS Fargate, NLB y API Gateway:

Listing 6: Despliegue de la infraestructura ECS

```
aws cloudformation create-stack \
  --stack-name ecs-stack-p1 \
  --template-body file://acoplada/config/ecs.yml \
  --parameters file://acoplada/config/ecs-params.json \
  --region $REGION

aws cloudformation wait stack-create-complete \
  --stack-name ecs-stack-p1 \
  --region $REGION
```

4.1.5. Obtención de Endpoints

Tras el despliegue exitoso, obtener los endpoints necesarios:

Listing 7: Obtención de la URL de la API

```
# URL base de API Gateway
aws cloudformation describe-stacks \
  --stack-name ecs-stack-p1 \
  --query "Stacks[0].Outputs[?OutputKey=='CharacterApiUrl'].OutputValue"
      \
  --output text

# ID de la API Key
aws cloudformation describe-stacks \
  --stack-name ecs-stack-p1 \
  --query "Stacks[0].Outputs[?OutputKey=='ApiKeyId'].OutputValue" \
  --output text

# Valor secreto de la API Key
aws apigateway get-api-key \
  --api-key <API_KEY_ID> \
  --include-value \
  --query 'value' \
  --output text
```

4.1.6. Limpieza de Recursos

Para eliminar todos los recursos creados:

Listing 8: Limpieza de recursos de la versión acoplada

```
# Eliminar pila ECS
aws cloudformation delete-stack --stack-name ecs-stack-p1 --region
    $REGION
aws cloudformation wait stack-delete-complete --stack-name ecs-stack-p1

# Eliminar pila DynamoDB
aws cloudformation delete-stack --stack-name bdd-stack-p1 --region
    $REGION
aws cloudformation wait stack-delete-complete --stack-name bdd-stack-p1

# Vaciar y eliminar repositorio ECR
aws ecr batch-delete-image \
  --repository-name characters-app \
  --image-ids "$(aws ecr list-images --repository-name characters-app \
    --query 'imageIds[*]' --output json --region $REGION)" \
  --region $REGION

aws cloudformation delete-stack --stack-name ecr-stack-p1 --region
    $REGION
aws cloudformation wait stack-delete-complete --stack-name ecr-stack-p1
```

4.2. Despliegue de la Versión Desacoplada (Serverless)

4.2.1. Prerrequisitos

Configurar las variables necesarias para el despliegue:

Listing 9: Configuración inicial para versión serverless

```
aws configure
export REGION='{TU_REGION}'
export ACCOUNT_ID='{TU_ID_DE_CUENTA_AWS}'
export S3_BUCKET_NAME='{TU_NOMBRE_DE_BUCKET_UNICO}'
export S3_KEY='lambdas-code/version-desacoplada.zip'

aws sts get-caller-identity
```

4.2.2. Despliegue de la Base de Datos

El proceso es idéntico al de la versión acoplada (utiliza la misma tabla DynamoDB).

4.2.3. Preparación del Paquete Lambda

Crear el bucket S3 y subir el paquete de código:

Listing 10: Subida del paquete Lambda a S3

```
# Crear bucket S3
aws s3 mb s3://$S3_BUCKET_NAME --region $REGION

# Subir paquete ZIP
aws s3 cp ./characters_lambda_package.zip \
  s3://$S3_BUCKET_NAME/$S3_KEY \
  --region $REGION
```

4.2.4. Despliegue de las Funciones Lambda

Desplegar la infraestructura serverless completa:

Listing 11: Despliegue de la infraestructura serverless

```
aws cloudformation create-stack \
  --stack-name serverless-crud-stack \
  --template-body file://desacoplada/config/lambdas.yml \
  --parameter-overrides file://lambdas-params.json \
  --region $REGION \
  --capabilities CAPABILITY_NAMED_IAM

aws cloudformation wait stack-create-complete \
  --stack-name serverless-crud-stack \
  --region $REGION
```

Nota: El parámetro `CAPABILITY_NAMED_IAM` es necesario porque la plantilla crea roles y políticas IAM.

4.2.5. Obtención de Endpoints

Listing 12: Obtención de endpoints serverless

```
# URL base de API Gateway
aws cloudformation describe-stacks \
  --stack-name serverless-crud-stack \
  --query "Stacks[0].Outputs[?OutputKey=='CharacterApiUrl'].OutputValue"
      \
  --output text

# Obtener API Key
API_KEY_ID=$(aws cloudformation describe-stacks \
  --stack-name serverless-crud-stack \
  --query "Stacks[0].Outputs[?OutputKey=='ApiKeyId'].OutputValue" \
  --output text)

aws apigateway get-api-key \
  --api-key $API_KEY_ID \
  --include-value \
  --query 'value' \
  --output text
```

4.2.6. Limpieza de Recursos

Listing 13: Limpieza de recursos serverless

```
# Eliminar pila Lambda
aws cloudformation delete-stack \
  --stack-name serverless-crud-stack \
  --region $REGION
aws cloudformation wait stack-delete-complete \
  --stack-name serverless-crud-stack

# Eliminar pila DynamoDB
aws cloudformation delete-stack \
  --stack-name bdd-stack-p1 \
  --region $REGION
aws cloudformation wait stack-delete-complete \
  --stack-name bdd-stack-p1

# Vaciar y eliminar bucket S3
aws s3 rm s3://$S3_BUCKET_NAME --recursive --region $REGION
aws s3 rb s3://$S3_BUCKET_NAME --force --region $REGION
```

5. Pruebas Funcionales

5.1. Pruebas Automáticas

Se proporciona un script Python para realizar pruebas automáticas del ciclo CRUD completo:

Listing 14: Ejecución de pruebas automáticas

```
python test/test_api_cycle.py
```

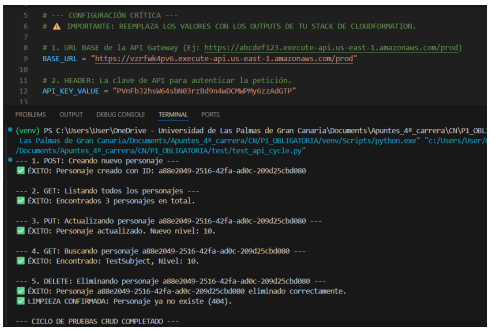

Figura 3: Resultado de la correcta ejecución del test python

Este script verifica secuencialmente los cinco endpoints establecidos (CREATE, READ, UPDATE, DELETE, LIST).

5.2. Pruebas Manuales

5.2.1. Colección Postman

Para realizar pruebas manuales con Postman:

1. Importar la colección `postman.json` en Postman.
2. Configurar la variable {{`ApiBaseUrl`}} con el valor de `CharacterApiUrl`.
3. Incluir el valor secreto de la API Key en el header `x-api-key` para todas las operaciones.

5.2.2. Interfaz Gráfica

Se proporciona una interfaz web básica ubicada en `frontend/frontend.html` que permite probar la API de forma visual e intuitiva. Deben introducirse los valores de la URL de la API y de la API KEY en la sección de configuración

Figura 4: Configuracion frontend

Figura 5: Interfaz gráfica

6. Análisis de Costos

6.1. Versión Monolítica Acoplada

Servicio	Descripción	Mes (USD)	Año (USD)
DynamoDB	Modo On-Demand (100k lecturas/escrituras)	0.02	0.24
ECR	Almacenamiento de imágenes Docker (0.65 GB)	0.07	0.84
ECS Fargate	2 tareas (0.25 vCPU, 0.5 GB RAM)	9.01	108.12
API Gateway	100k llamadas API REST	0.35	4.20
NLB	Balanceo de carga interno	16.47	197.64
Total	Entorno de desarrollo/bajo tráfico	**25.87**	**310.47**

Cuadro 4: Estimación de costos - Versión Acoplada

6.2. Versión Desacoplada Serverless

Servicio	Descripción	Mes (USD)	Año (USD)
DynamoDB	Modo On-Demand (100k lecturas/escrituras)	0.02	0.24
S3	Almacenamiento paquete Lambda (∼50 MB)	0.02	0.24
Lambda	Ejecución (1 tarea, 0.25 vCPU, 0.5 GB RAM)	0.01	0.12
API Gateway	100k llamadas API REST	0.35	4.20
Total	Entorno de desarrollo/bajo tráfico	**0.45**	**5.45**

Cuadro 5: Estimación de costos - Versión Serverless

6.3. Comparativa de Costos

La versión serverless presenta un ahorro significativo:

- **Ahorro mensual**: USD 25.42 (98.3 % de reducción)
- **Ahorro anual**: USD 305.02 (98.2 % de reducción)

Este ahorro se debe principalmente a:

- Ausencia de Network Load Balancer
- Modelo de pago por uso de Lambda vs. contenedores siempre activos en ECS
- Menores costos de almacenamiento (S3 vs. ECR)

7. Conclusiones

7.1. Ventajas y Desventajas de Cada Arquitectura

7.1.1. Versión Monolítica Acoplada (ECS Fargate)

Ventajas:

- Latencia más predecible (contenedores siempre activos)
- Mayor control sobre el entorno de ejecución
- Mejor para cargas de trabajo constantes
- Facilita la depuración y el monitoreo centralizado

Desventajas:

- Costos fijos independientemente del uso
- Mayor complejidad de infraestructura
- Requiere gestión de escalado
- Costos significativamente superiores para bajo tráfico

7.1.2. Versión Desacoplada Serverless (Lambda)

Ventajas:

- Modelo de pago por uso real
- Escalado automático sin configuración
- Costos extremadamente reducidos para bajo tráfico
- Menor complejidad operacional
- Arquitectura completamente serverless

Desventajas:

- Cold starts pueden aumentar la latencia
- Límites de ejecución y memoria de Lambda
- Complejidad en la gestión de múltiples funciones
- Costos pueden aumentar significativamente con alto tráfico constante

7.2. Recomendaciones de Uso

- **Elegir ECS Fargate cuando**:
  - Se requiere latencia baja y predecible
  - El tráfico es constante y elevado
  - Se necesita mayor control del entorno
  - La aplicación requiere procesos de larga duración
- **Elegir Lambda cuando**:
  - El tráfico es variable o bajo
  - Se prioriza la reducción de costos
  - La aplicación puede dividirse en funciones independientes
  - No se requieren procesos de más de 15 minutos

7.3. Conclusión Final

Ambas arquitecturas son válidas y funcionales para el caso de uso propuesto. La elección entre una u otra depende fundamentalmente de:

1. Patrones de tráfico esperados
2. Requisitos de latencia
3. Presupuesto disponible
4. Experiencia del equipo de desarrollo
5. Necesidades de escalabilidad

Para el contexto de esta práctica (entorno de desarrollo con bajo tráfico), la arquitectura serverless resulta significativamente más económica y sencilla de gestionar, con un ahorro de aproximadamente 98 % en costos operacionales.