CLOTHING WISHLIST API
Memoria Entregable 1

Computación en la Nube

WAIL BEN EL HASSANE BOUDHAR

Universidad de Las Palmas de Gran Canaria

1. Introducción

1.1. Objetivo del proyecto

El objetivo principal del proyecto **Clothing Wishlist API** es diseñar y desplegar de dos modos distintos una **API RESTful con operaciones CRUD** (Create, Read, Update, Delete) que permita gestionar una lista de deseos (“wishlist”) de prendas de ropa con la finalidad de hacer la comparativa entre ambos versionados.

La aplicación persiste los datos en **Amazon DynamoDB**, un servicio de base de datos NoSQL gestionado por AWS, y se despliega utilizando dos enfoques de arquitectura distintos:

- **Versión acoplada:** basada en contenedores **ECS Fargate**, con balanceo de carga mediante **Network Load Balancer (NLB)** y exposición de la API a través de **API Gateway** mediante un **VPC Link**.

- **Versión desacoplada:** basada en **funciones AWS Lambda** empaquetadas en una imagen de contenedor almacenada en **Elastic Container Registry (ECR)** e integradas con **API Gateway** en modo proxy.

Ambos despliegues buscan demostrar el conocimiento de **diseño, desacoplamiento, despliegue y escalabilidad en entornos cloud**, aplicando servicios fundamentales de AWS para lograr una aplicación robusta, mantenible y fácilmente extensible.

1.2. Descripción general de la aplicación

La **Clothing Wishlist API** expone un conjunto de endpoints HTTP bajo el recurso principal /clothing, permitiendo crear, consultar, actualizar y eliminar elementos.

Entre las rutas implementadas destacan:

- **POST /clothing** – Crea un nuevo elemento.
- **GET /clothing** – Lista todos los elementos.
- **GET /clothing/{id}** – Recupera un elemento específico por ID.
- **PUT /clothing/{id}** – Actualiza una prenda existente.
- **DELETE /clothing/{id}** – Elimina una prenda de la lista.

Además, la aplicación implementa un **endpoint de verificación de estado** (/health) y soporta **CORS**, lo que permite su uso desde un cliente web incluido en el proyecto (web/), facilitando las pruebas manuales.

Las operaciones CRUD se integran con DynamoDB a través de un repositorio especializado que gestiona la conexión y la serialización de los objetos de tipo “clothing”.

1.3. Tecnologías y servicios utilizados

El proyecto hace uso de los siguientes **servicios y herramientas de AWS**:

- **Amazon DynamoDB:** almacenamiento NoSQL de las prendas de la wishlist.
- **Amazon ECS Fargate:** ejecución sin servidor de contenedores para la versión acoplada.
- **AWS Lambda:** ejecución serverless de funciones en contenedor para la versión desacoplada.
- **Amazon API Gateway:** gestión y publicación de la API en ambas versiones.
- **Amazon ECR (Elastic Container Registry):** repositorio de imágenes Docker para los despliegues.
- **AWS CloudFormation:** definición de la infraestructura mediante plantillas YAML reproducibles.
- **Amazon CloudWatch Logs:** registro y monitorización de logs de ejecución y eventos del sistema.
- **IAM (Identity and Access Management):** control de permisos y roles para los distintos recursos.
- **Network Load Balancer (NLB):** balanceo interno del tráfico en la arquitectura ECS.

A nivel de desarrollo, la API está construida con **Node.js y Express**, y los contenedores se gestionan mediante **Docker**. Se incluye un script PowerShell (create-image.ps1) para automatizar la construcción y publicación de las imágenes en ECR.

2. Diseño de la Arquitectura

2.1. Visión general

El proyecto se ha diseñado siguiendo principios de **modularidad, escalabilidad y resiliencia** en entornos de computación en la nube.
Para cumplir los objetivos del laboratorio, se implementaron **dos versiones arquitectónicas complementarias** que muestran diferentes enfoques de despliegue en AWS:

- **Versión acoplada (ECS Fargate):**
  Se basa en una arquitectura de contenedores gestionados por AWS Fargate, con un **Network Load Balancer (NLB)** interno que distribuye las peticiones hacia las tareas en ejecución dentro de un **ECS Cluster**. La comunicación externa se realiza a través de **API Gateway**, que accede al NLB mediante un **VPC Link**.

- **Versión desacoplada (Lambda):**
  Se fundamenta en un enfoque **serverless**, donde cada operación CRUD se implementa como una **función Lambda** independiente. Las funciones se empaquetan como una **imagen Docker alojada en Amazon ECR**, y la exposición pública se realiza a través de **API Gateway** en modo **AWS_PROXY**, que reenvía directamente las solicitudes a cada función.

Ambas versiones utilizan **Amazon DynamoDB** como base de datos NoSQL para almacenar los elementos de la lista de deseos de prendas.

El diseño busca mostrar las ventajas y compromisos entre **una arquitectura tradicional de servicios en contenedor (más control, mayor gestión)** y **una arquitectura serverless (más agilidad, menor coste operativo)**.

2.2. Arquitectura Acoplada – ECS Fargate

En la versión acoplada, la aplicación Node.js (basada en Express) se despliega dentro de un **contenedor Docker** ejecutado en **ECS Fargate**.

El flujo general de petición es el siguiente:

1. El cliente realiza una petición HTTPS a **API Gateway**.
2. API Gateway válida la **API Key** y redirige la solicitud mediante un **VPC Link** hacia el **Network Load Balancer (NLB)**.
3. El NLB distribuye el tráfico hacia el **servicio ECS**, que contiene una o más **tareas Fargate** ejecutando la aplicación Express.
4. La aplicación procesa la operación solicitada (CRUD) y accede a la tabla DynamoDB configurada mediante variables de entorno.
5. La respuesta se devuelve al cliente a través del mismo flujo inverso (ECS → NLB → API Gateway → cliente).

Componentes principales:

- **ECS Cluster y Service:** gestionan la ejecución de contenedores.
- **NLB interno:** balancea tráfico HTTP en el puerto 8080 y realiza health checks sobre /health.
- **API Gateway (con VPC Link):** expone la API de forma controlada sin exponer directamente la VPC.
- **DynamoDB:** almacenamiento persistente de los elementos de la wishlist.
- **CloudWatch Logs:** registro de logs de aplicación y métricas de ECS.

2.3. Arquitectura Desacoplada – AWS Lambda

La versión desacoplada se implementa mediante **cinco funciones AWS Lambda**, cada una representando una operación CRUD.

Cada función se despliega a partir de una **misma imagen ECR**, que contiene un dispatcher interno encargado de seleccionar el handler correspondiente según la variable HANDLER_NAME.

El flujo de petición es el siguiente:

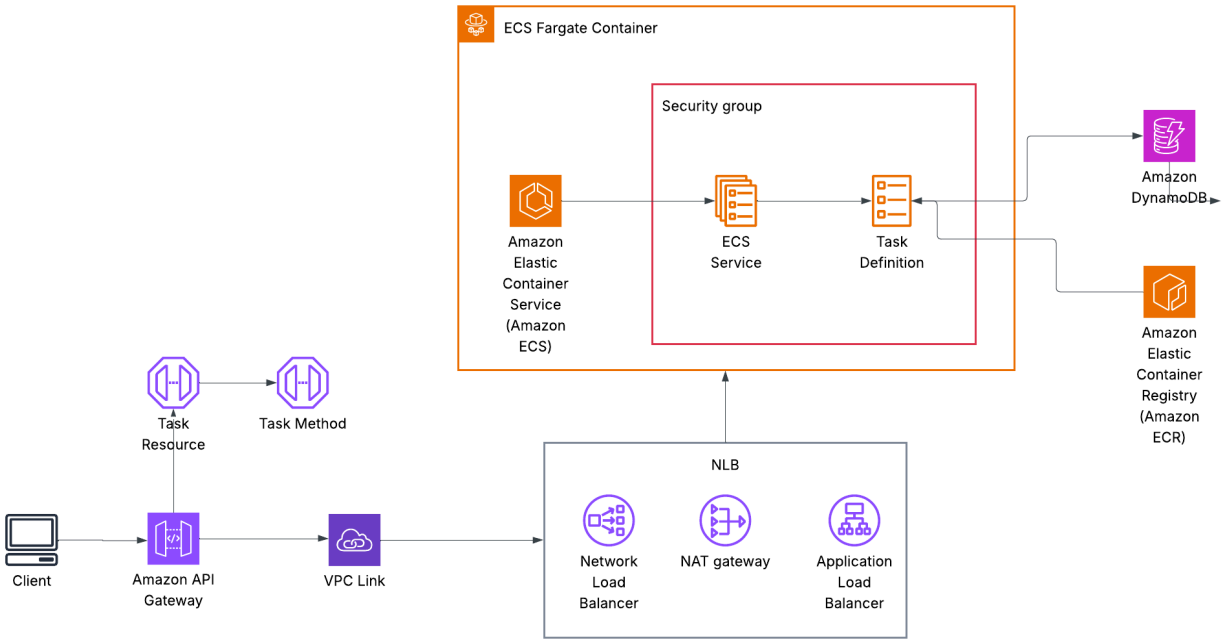
1. El cliente envía una solicitud a **API Gateway**.
2. API Gateway valida la **API Key** y reenvía la petición directamente (modo **AWS_PROXY**) a la función Lambda correspondiente.
3. La función ejecuta la operación solicitada sobre la tabla **DynamoDB** utilizando el SDK de AWS.
4. El resultado se devuelve directamente a API Gateway, que responde al cliente.

Componentes principales:

- **AWS Lambda:** 5 funciones (Get, List, Post, Put, Delete) empaquetadas en imagen ECR.
- **Amazon ECR:** repositorio de la imagen Docker compartida.
- **API Gateway:** integra con cada función Lambda y gestiona CORS, documentación y API Keys.
- **DynamoDB:** base de datos NoSQL compartida entre todas las funciones.
- **CloudWatch Logs:** monitorización y trazabilidad de invocaciones

2.4. Diagrama conceptual

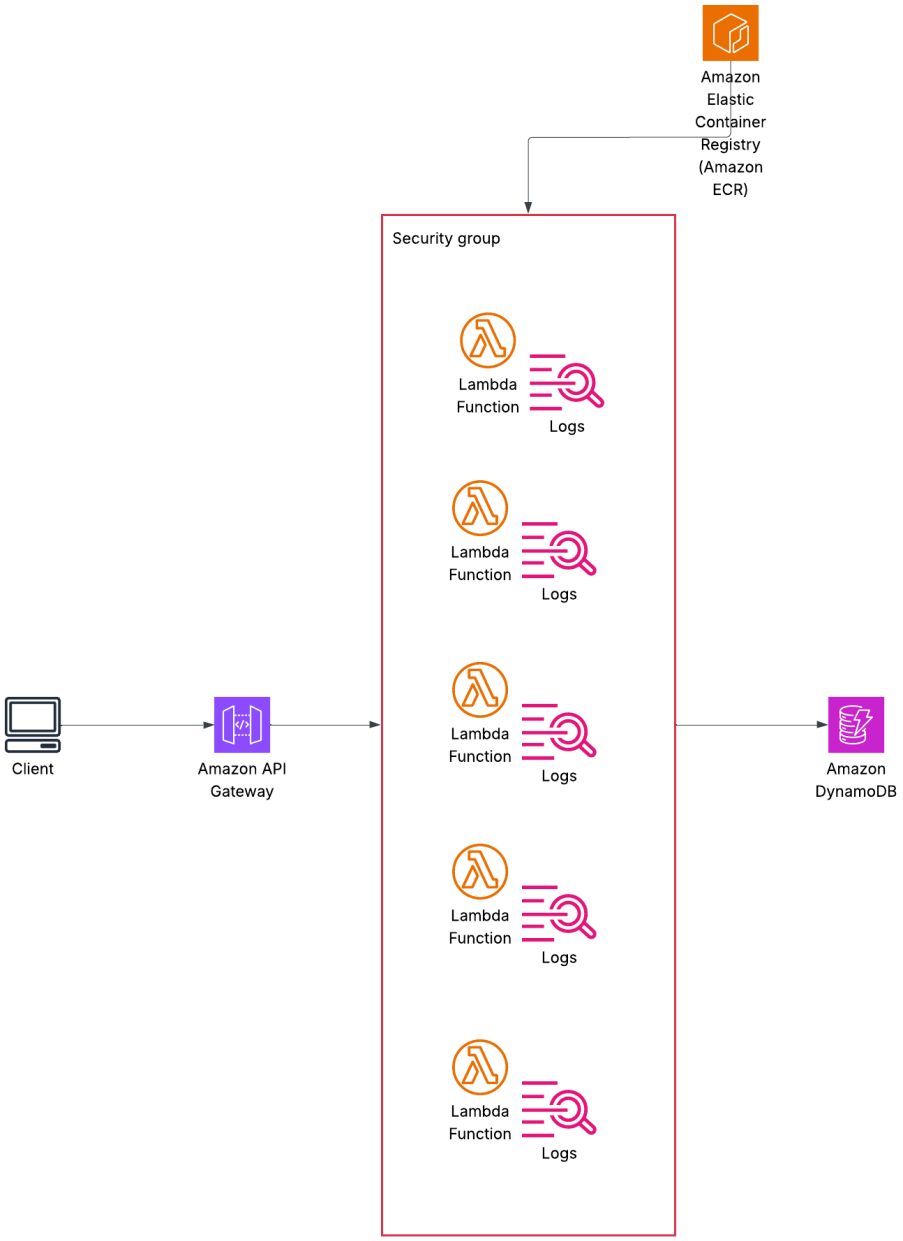
Acoplada:



Características clave:

- NLB interno (no expuesto públicamente).
- API Gateway como único punto de entrada seguro.
- VPC Link para mantener la comunicación dentro de la VPC.

Desacoplada:



Características clave:

- Arquitectura completamente serverless.
- Alta escalabilidad y bajo mantenimiento.
- Coste basado en uso real (pago por invocación y consumo de recursos).

2.5. Justificación del diseño

El diseño de ambas versiones se ha guiado por los siguientes criterios:

- **Escalabilidad:** ambas arquitecturas escalan horizontalmente según la carga (tareas Fargate o invocaciones Lambda).
- **Resiliencia:** uso de servicios gestionados reduce puntos únicos de fallo.
- **Seguridad:** todo el tráfico externo pasa por API Gateway; el NLB es interno; se emplea el rol (labRole).
- **Desacoplamiento:** la versión Lambda separa completamente las funciones de negocio, permitiendo actualizaciones independientes.
- **Costo y simplicidad operativa:** DynamoDB en modo *PAY_PER_REQUEST* y Lambda reducen costes y mantenimiento para entornos de laboratorio.

3. Implementación

3.1. Diseño de la Base de Datos

Se ha definido una **tabla gestionada denominada clothing_items**, desplegada mediante una plantilla **CloudFormation**, siguiendo el principio *Infrastructure as Code (IaC)*.

Cada elemento de la tabla representa una **prenda de ropa** dentro de la lista de deseos del usuario.

La estructura definida en el YAML es la siguiente:

- **Clave primaria (Partition Key):** id *(tipo String)*.
- **Índice Secundario Global (GSI):** name-index, con el atributo name como clave de partición, permitiendo realizar búsquedas por nombre de prenda.
- **Modo de facturación:** PAY_PER_REQUEST, lo que elimina la necesidad de definir capacidad provisionada y optimiza costes para entornos con carga variable o baja.

Atributos principales por registro:

Atributo	Tipo	Descripción
id	String	Identificador único (UUID)
name	String	Nombre de la prenda
brand	String	Marca de la prenda
size	String	Talla
color	String	Color
price	Number	Precio estimado
wishlist	Boolean	Indicador de si está marcada como deseada
notes	String	Observaciones opcionales
createdAt	String	Fecha de creación
updatedAt	String	Fecha de última actualización

El acceso a DynamoDB se realiza mediante el cliente **AWS SDK v3** (@aws-sdk/lib-dynamodb), utilizando el patrón **Repository** para desacoplar la capa de datos de la lógica de negocio.

Este repositorio (DynamoDBClothingRepository) implementa las operaciones CRUD mediante comandos específicos de la API de documentos (PutCommand, GetCommand, UpdateCommand, DeleteCommand, ScanCommand).

3.2. Implementación de la API y Operaciones CRUD

La aplicación expone un conjunto de endpoints HTTP RESTful bajo el recurso base /clothing, implementados con **Express.js** y desplegados tanto en **ECS Fargate** como en **AWS Lambda** (según la versión de arquitectura).

Cada ruta está asociada a un controlador que delega la lógica de acceso a la base de datos en el repositorio mencionado.
El formato de las respuestas es **JSON estructurado**, con la siguiente convención:

```
{
 "success": true,
 "data": { ... },
 "message": "Operation completed successfully"}
```

Endpoints principales:

Método	Ruta	Descripción	Estado HTTP esperado
POST	/clothing	Crea una nueva prenda	201 Created
GET	/clothing	Obtiene todas las prendas	200 OK
GET	/clothing/{id}	Recupera una prenda por su ID	200 OK / 404 Not Found
PUT	/clothing/{id}	Actualiza una prenda existente	200 OK
DELETE	/clothing/{id}	Elimina una prenda	200 OK / 404 Not Found

El controlador incluye validaciones básicas sobre los datos de entrada, como la presencia de campos obligatorios (name, brand, size, color, price) y la comprobación de tipos para los valores numéricos y booleanos.

El endpoint /health se utiliza para verificar el estado del servicio y es consumido tanto por el balanceador de carga (**NLB**) como por el frontend web de pruebas.

3.3. Desacoplamiento y Lógica Interna

El diseño de la aplicación sigue un **modelo por capas** que permite aislar la lógica de negocio de la infraestructura.

Las principales capas son:

- **Capa de Rutas (Express / API Gateway):** recibe las peticiones HTTP y las direcciona al controlador correspondiente.
- **Capa de Controladores:** implementa la lógica CRUD, validando datos y gestionando errores.
- **Capa de Repositorio (DynamoDB):** abstrae el acceso a la base de datos mediante el patrón Repository.
- **Capa de Infraestructura (AWS):** definida mediante CloudFormation YAML, desplegando de forma declarativa la API, las funciones Lambda o el servicio ECS.

En la **versión Lambda**, cada operación CRUD se empaqueta como una función independiente que se ejecuta en contenedor desde una imagen alojada en **Amazon ECR**.
Estas funciones operan de forma **totalmente desacoplada**, compartiendo únicamente la tabla DynamoDB y el código base común dentro de la imagen.

Este enfoque permite un escalado automático e independiente para cada operación según la demanda.

En la **versión ECS**, la aplicación Express se ejecuta como un servicio monolítico dentro de un contenedor gestionado por **Fargate**. Aunque todos los endpoints conviven dentro de una misma instancia, el desacoplamiento se logra mediante la separación lógica del código y la abstracción del acceso a datos a través del repositorio.

3.4. Seguridad y Control de Acceso

La seguridad del despliegue se basa en tres niveles:

1. **Autenticación mediante API Key:**
   Ambos despliegues (ECS y Lambda) están protegidos mediante **API Gateway con API Key obligatoria**, lo que restringe el acceso únicamente a clientes autorizados.

2. **Control de permisos IAM:**
   Tanto las funciones Lambda como las tareas ECS utilizan el rol gestionado LabRole, que otorga permisos controlados para acceder a DynamoDB, CloudWatch Logs y ECR.

3. **Red interna privada (VPC):**
   En la versión ECS, el **Network Load Balancer (NLB)** está configurado en modo **interno**, y la comunicación entre API Gateway y ECS se realiza mediante un **VPC Link**, garantizando que las peticiones nunca salgan a Internet público.

Además, todas las solicitudes se procesan sobre **HTTPS**, y las políticas CORS se configuran en API Gateway para permitir únicamente los métodos y cabeceras necesarios (GET, POST, PUT, DELETE, OPTIONS).

3.5. Cliente Web de Pruebas

Como parte del proyecto, se ha desarrollado una **interfaz web rudimentaria** para realizar pruebas sobre la API.

Esta interfaz permite configurar la URL base y la API Key, visualizar la lista de prendas, añadir, editar o eliminar elementos, y comprobar el estado del servidor mediante el endpoint /health.

El cliente está desarrollado en **JavaScript nativo** y funciona completamente desde el navegador, facilitando la demostración y validación de las operaciones CRUD de forma interactiva.

4. Despliegue y Pruebas

4.1. Proceso de Despliegue

El proyecto se desplegó completamente sobre **Amazon Web Services (AWS)** utilizando una combinación de infraestructura como código (IaC) mediante **CloudFormation** y recursos definidos en archivos **YAML**.

Para la construcción y publicación de las imágenes de contenedor, se empleó un **script PowerShell automatizado** que simplifica el flujo de trabajo de construcción, etiquetado y subida a **Amazon Elastic Container Registry (ECR)**.

4.2. Construcción y publicación de la imagen Docker

El script create-image.ps1 automatiza el proceso de generación de la imagen de la aplicación y su publicación en el repositorio de ECR.

El flujo ejecutado por el script es el siguiente:

Inicio de sesión en ECR
Utiliza el comando aws ecr get-login-password para autenticar Docker con el registro privado de ECR:

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 058264528120.dkr.ecr.us-east-1.amazonaws.com

1. **Construcción de la imagen**
   La imagen se construye a partir del contexto del proyecto (o un Dockerfile específico, si se indica):

   docker build --platform linux/amd64 --provenance=false -t $EcrName $Path

2. **Etiquetado y publicación**
   Una vez construida, se etiqueta la imagen con el repositorio remoto y se sube a ECR:

   docker tag
   ${EcrName}:latest058264528120.dkr.ecr.us-east-1.amazonaws.com/${EcrName}:latest

   docker push 058264528120.dkr.ecr.us-east-1.amazonaws.com/${EcrName}:latest

3. El script incluye validaciones básicas, control de errores y compatibilidad con rutas personalizadas para el Dockerfile.

De esta forma, se garantiza que tanto la versión ECS como la Lambda utilicen una **imagen consistente y verificada**.

4.3. Despliegue de infraestructura

El despliegue de los recursos de AWS (ECS, Lambda, API Gateway, DynamoDB y VPC Link) se realizó desde la **consola gráfica de AWS CloudFormation**, seleccionando las plantillas YAML correspondientes:

- ecs-deployment.yaml → Despliega la versión acoplada con ECS Fargate.
- lambda-deployment.yaml → Despliega la versión desacoplada con funciones Lambda.
- dynamo.yaml → Crea la tabla DynamoDB con su índice global secundario.

No obstante, también es posible realizar el despliegue completo mediante la **AWS CLI**, ejecutando el siguiente comando desde un entorno local configurado con credenciales válidas:

```
aws cloudformation deploy \
  --template-file lambda-deployment.yaml \
  --stack-name clothing-lambda-stack \
  --capabilities CAPABILITY_IAM
```

Y de forma equivalente para la versión ECS:

```
aws cloudformation deploy \
  --template-file ecs-deployment.yaml \
  --stack-name clothing-ecs-stack \
  --capabilities CAPABILITY_IAM \
  --parameter-overrides ImageName=clothing-app:latest \
                        DBDynamoName=clothing_items \
                        VpcId=vpc-XXXXXXXX \
                        SubnetIds='["subnet-XXXX","subnet-YYYY"]'
```

Ambos despliegues generan como **salida (Outputs)** la URL base del API Gateway y la **API Key** necesaria para consumir los endpoints.

4.4. Verificación funcional

La validación del correcto funcionamiento de la aplicación se llevó a cabo utilizando una **interfaz web de pruebas** incluida en el proyecto.
Esta interfaz (ubicada en la carpeta web/) permite:

- Configurar la **URL del API Gateway** y la **API Key**.
- Visualizar todas las prendas almacenadas (GET /clothing).
- Crear nuevas prendas (POST /clothing).
- Editar o eliminar prendas existentes (PUT y DELETE /clothing/{id}).
- Verificar el estado del servicio mediante el endpoint /health.

La interfaz implementa llamadas **fetch()** a la API REST y muestra los resultados en tiempo real, lo que facilita la verificación de las operaciones CRUD sin necesidad de herramientas externas como Postman.

Durante las pruebas, se validaron correctamente los siguientes aspectos:

- Inserción, consulta, actualización y eliminación de prendas.
- Persistencia de los datos en la tabla DynamoDB.
- Manejo de errores (campos obligatorios, elementos inexistentes, etc.).
- Comprobación de CORS y headers de seguridad en el navegador.
- Funcionamiento del endpoint /health tanto en ECS como en Lambda.

4.5. Supervisión y Logs

Durante el despliegue y las pruebas, se utilizó **Amazon CloudWatch Logs** para la monitorización de las funciones y servicios:

- **Lambda Functions:** generaron logs automáticos bajo /aws/lambda/<function-name>, permitiendo analizar cada invocación.
- **ECS Fargate:** los contenedores registraron su salida estándar bajo /ecs/clothing, accesible desde la consola de CloudWatch.

Esta monitorización permitió comprobar la correcta conexión con DynamoDB, la invocación de las funciones Lambda y el estado de salud del servicio en ECS.

4.6. Resultados del Despliegue

El despliegue final resultó exitoso en ambos entornos:

- **Versión ECS Fargate:** servicio disponible en red privada a través de **API Gateway + VPC Link**.

- **Versión Lambda:** endpoints accesibles directamente desde API Gateway, con generación automática de documentación.

- **Base de datos DynamoDB:** tabla operativa con GSI en name, utilizada correctamente por todas las operaciones CRUD.

- **Frontend web:** interfaz funcional para pruebas en vivo, sin necesidad de herramientas externas.

5. Análisis de Costes (Pricing)

5.1. Enfoque del análisis

El análisis de costes compara las dos arquitecturas implementadas para la aplicación **Clothing Wishlist API**, desplegadas en la región **us-east-1 (N. Virginia)** bajo un escenario de producción **24/7**, con una carga moderada de uso:

- **Invocaciones Lambda mensuales:** 100 000
- **Llamadas API Gateway mensuales:** 100 000
- **Lecturas/escrituras DynamoDB mensuales:** 1 000 000
- **Tarea ECS Fargate permanente:** 1 tarea (CPU: 256, Memoria: 512 MiB)

Se asume que los recursos están desplegados con configuraciones por defecto y sin descuentos por reserva.

5.2. Coste estimado — Arquitectura Acoplada (ECS Fargate)

Total mensual: ≈ 42,23 USD
Total anual: ≈ 506,76 USD

Desglose principal (mensual):

- **Network Load Balancer: 16,44 USD**
- **AWS PrivateLink (VPC Link): 14,61 USD**
- **AWS Fargate: 9,01 USD**
- Amazon CloudWatch: **1,01 USD**
- DynamoDB on-demand capacity: **0,69 USD**
- Amazon API Gateway (REST): **0,35 USD**
- Data Transfer: **0,11 USD**
- Amazon ECR: **0,01 USD**

5.3. Coste estimado — Arquitectura Desacoplada (Lambda)

Total mensual: ≈ 1,31 USD
Total anual: ≈ 15,77 USD

Desglose principal (mensual):

- Amazon CloudWatch: **0,5045 USD**
- Amazon API Gateway (REST): **0,35 USD**
- DynamoDB on-demand capacity: **0,28 USD**
- AWS Lambda: **0,18 USD**

5.4. Comparativa

Concepto	Desacoplada (Lambda)	Acoplada (ECS)	Diferencia
Coste mensual	1,31 USD	42,23 USD	–96,9%
Coste anual	15,77 USD	506,76 USD	–491,0 USD/año

Conclusión de la tabla:

Para el patrón de uso asumido, la **versión desacoplada (Lambda)** es **claramente más eficiente en coste**. En la arquitectura acoplada, el gasto lo dominan **NLB** y **PrivateLink (VPC Link)**, que introducen un coste fijo mensual relevante incluso con tráfico moderado. En la arquitectura desacoplada, el coste está altamente ligado al **uso real** (invocaciones y logs), manteniéndose muy bajo.

6.Conclusión

En términos generales, **una arquitectura acoplada (basada en contenedores o instancias ECS/EC2)** suele ser más conveniente cuando la aplicación:

- Requiere **procesos en ejecución continua**, por ejemplo, servidores web que mantienen sesiones persistentes, colas internas o tareas de fondo.
- Necesita **mayor control sobre el entorno de ejecución**, versiones del sistema operativo o dependencias específicas.
- Debe manejar **volúmenes de tráfico constantes o predecibles**, donde el coste fijo se compensa con una utilización continua de los recursos.
- Implica **procesamiento de alto rendimiento o cálculos intensivos**, que podrían verse limitados por los tiempos de ejecución o las cuotas de Lambda.

Por el contrario, **una arquitectura desacoplada (basada en AWS Lambda)** resulta más ventajosa en proyectos donde:

- Las cargas son **intermitentes o variables**, ya que se paga solo por invocación y tiempo de ejecución.
- Se busca **reducir costes de infraestructura** y simplificar el mantenimiento operativo.
- La aplicación puede dividirse fácilmente en funciones independientes (CRUD, validaciones, eventos).
- Se prioriza la **escalabilidad automática y alta disponibilidad sin gestión de servidores**.

En este proyecto, la aplicación no requiere ejecución continua ni conexiones persistentes, y su carga de trabajo se ajusta perfectamente al modelo de **funciones por demanda**.

Por ello, la **arquitectura desacoplada** no solo ofrece un **ahorro aproximado del 97 %** respecto a la versión acoplada, sino que también reduce significativamente la complejidad operativa, el mantenimiento y la necesidad de supervisión de infraestructura, manteniendo idéntica funcionalidad y disponibilidad.