DISEÑO DE APLICACIONES EN LA NUBE

AP1: Diseño de aplicaciones básicas en la nube

10 DE NOVIEMBRE DE 2025

ESTEBAN TRUJILLO SANTANA
DNI: 45346949

INDICE

MEMORIA TÉCNICA Y DEFENSA DEL PROYECTO CLOUD

1. INTRODUCCIÓN Y OBJETIVOS DEL PROYECTO

1.1. Contexto y Requisitos

En este proyecto se aborda el diseño y la implementación de una API REST para la gestión de recursos genéricos (libros/books) en la plataforma Amazon Web Services (AWS). El objetivo principal es la creación de una solución de *backend* que cumpla con los estándares de robustez, escalabilidad y seguridad requeridos en entornos de producción en la nube.

Definición del Proyecto (API CRUD)

La funcionalidad esencial de la API se centra en la provisión de las cuatro operaciones fundamentales de gestión de datos, conocidas como CRUD:

- **POST (Create):** Creación de nuevos elementos.
- **GET (Read):** Consulta de todos los elementos o de un elemento específico mediante su ID.
- **PUT (Update):** Modificación de los datos de un elemento existente.
- **DELETE (Delete):** Eliminación de un elemento por su ID.

Objetivo Principal (Doble Arquitectura y Comparativa)

El objetivo central es demostrar el dominio en la implementación de la misma lógica de negocio bajo dos paradigmas arquitectónicos opuestos:

1. **Arquitectura Acoplada (o Monolítica):** Basada en la contenerización de la aplicación (Docker) ejecutada en el servicio de orquestación de AWS, ECS Fargate.
2. **Arquitectura Desacoplada (o Serverless):** Basada en el despliegue de funciones bajo demanda mediante AWS Lambda.

Esta dualidad permite una comparativa directa en la defensa del proyecto, analizando las implicaciones en términos de coste operativo (TCO), escalabilidad horizontal y modelo de gestión de infraestructura.

1.2. Requisitos Técnicos Obligatorios

El diseño del proyecto está condicionado por los siguientes requisitos tecnológicos mandatorios:

Base de Datos: **DynamoDB**

Se requiere el uso de AWS DynamoDB, una base de datos NoSQL de tipo clave-valor. La elección se fundamenta en su capacidad para ofrecer:

- **Escalabilidad Ilimitada:** DynamoDB gestiona automáticamente el tráfico sin necesidad de aprovisionar capacidad manualmente.
- **Alto Rendimiento:** Garantiza una latencia mínima y predecible, crucial para aplicaciones de alta demanda.
- **Modelo de Coste (Pay-per-Request):** El modo PAY_PER_REQUEST se alinea perfectamente con la arquitectura Serverless, ya que solo se consume capacidad y se incurre en coste cuando se realizan transacciones de lectura o escritura.

Operaciones CRUD y Endpoints HTTP

La API se expone con los siguientes *endpoints*, verificados mediante la colección de pruebas de Postman:

Endpoint HTTP	Método	Operación CRUD	Descripción Técnica
/items	POST	Create	Inserción de un nuevo ítem en la tabla.
/items/{id}	GET	Read (by ID)	Consulta del ítem mediante su clave primaria.
/items	GET	Read (All)	Retorno de la lista completa de ítems.
/items/{id}	PUT	Update	Actualización parcial o total del registro.
/items/{id}	DELETE	Delete	Eliminación del registro por ID.

Interfaz de Acceso: API Gateway

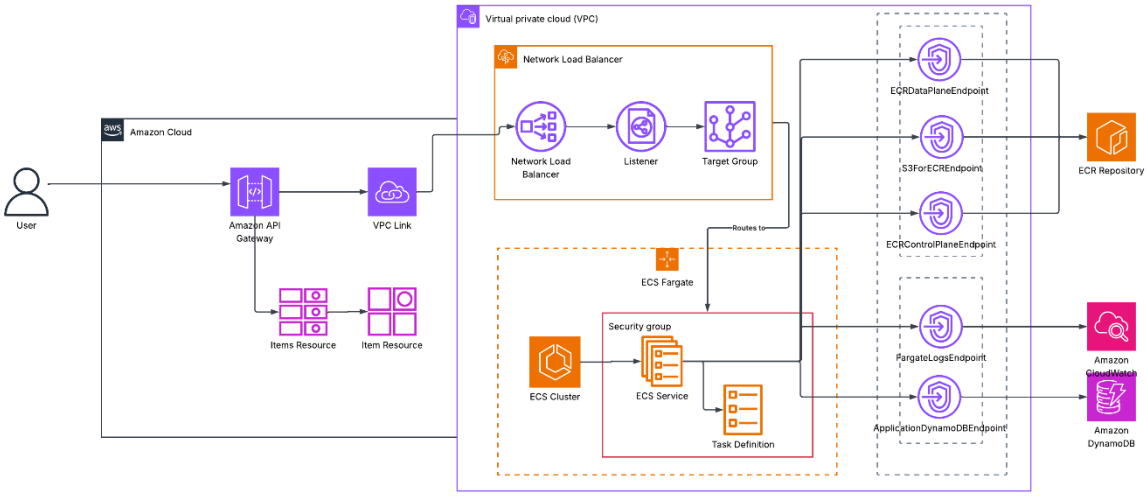
El servicio AWS API Gateway es el punto de entrada unificado y seguro para ambas arquitecturas. Sus funciones principales son:

- **Seguridad y Control de Acceso:** Centraliza la validación de la API Key (x-api-key) en la cabecera de la petición, protegiendo los servicios de *backend* del tráfico no autorizado.
- **Orquestación:** Actúa como *router* o fachada, desacoplando al cliente de la lógica interna. En la arquitectura Acoplada, utiliza un *VPC Link* para enlazar con el balanceador de carga; en la Desacoplada, realiza la integración directa con la función Lambda correspondiente.

2. DISEÑO Y ARQUITECTURAS DE LA SOLUCIÓN

Esta sección compara dos implementaciones distintas para la API CRUD, ambas aisladas a nivel de infraestructura: cada una utiliza su propio API Gateway, API Key y tabla DynamoDB paralela.

2.1. Arquitectura 1: Acoplada (ECS Fargate)



Representa un diseño monolítico tradicional. La aplicación Python/Flask se empaqueta en un contenedor Docker y se ejecuta de forma persistente.

Flujo y Componentes Clave

Componente	**Función Principal**	**Flujo de la Petición (POST /items)**
API Gateway 1	Fachada pública y Validador de API Key 1.	1. Recibe la petición del cliente. 2. Valida la clave.
VPC Link	Enlace privado (API Gateway → VPC).	3. Enruta el tráfico al NLB (evita exponer la red privada).
Network Load Balancer (NLB)	Distribuidor de tráfico.	4. Reparte la carga a una tarea activa de ECS.
ECS Fargate	Entorno de Cómputo (Contenedor Flask).	5. El contenedor Flask procesa la petición y llama a DynamoDB.

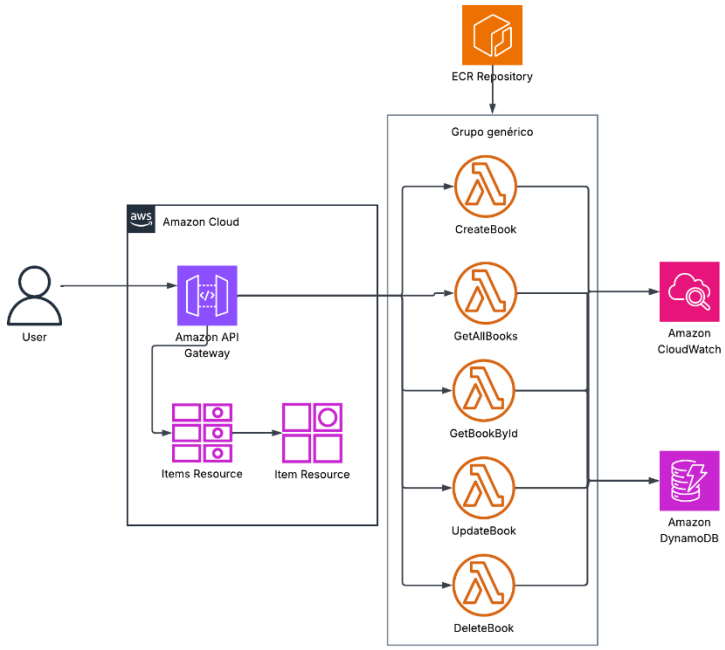
DynamoDB (Tabla 1)	Persistencia de datos (acceso privado por VPC Endpoint).	6. Guarda el ítem. Respuesta en sentido inverso.

Ventaja Clave: Control total sobre el entorno y estado de la aplicación (sesiones, caché interna).

Desventaja Clave: Mayor coste y complejidad en la gestión de red (NLB, VPC Link).

El flujo detallado para la operación POST es análogo para los demás *endpoints* (GET, PUT, DELETE); la única diferencia radica en la acción ejecutada por el contenedor Flask y el método de la base de datos (p. ej., delete_item en lugar de put_item).

2.2. Arquitectura 2: Desacoplada (AWS Lambda)



Representa un diseño *Serverless* moderno. La lógica se descompone en cinco funciones Lambda, una por cada *endpoint* CRUD.

Flujo y Componentes Clave

Componente	**Función Principal**	**Flujo de la Petición (POST /items)**
API Gateway 2	Fachada pública y Router de Funciones.	1. Recibe la petición y valida la API Key 2.

Lambda Functions	Cómputo efímero (5 funciones).	2. API Gateway invoca directamente la función create_item_handler.
Rol de Ejecución (IAM)	Permisos de seguridad.	3. Proporciona las credenciales temporales a la función (ej. LabRole) para acceder a DynamoDB, aplicando el principio de mínimos privilegios.
DynamoDB (Tabla 2)	Persistencia de datos.	4. La Lambda ejecuta put_item en la tabla. 5. Respuesta en sentido inverso.

Ventaja Clave: Modelo de Pago por Uso (cero costes en inactividad) y escalabilidad automática (por petición).

Desventaja Clave: Latencia de inicio ("Cold Start") y la necesidad de gestionar permisos IAM para cada función.

El flujo de los demás *endpoints* (GET, PUT, DELETE) es análogo al POST; la diferencia reside únicamente en la acción ejecutada por la función Lambda específica y el método invocado en DynamoDB (p. ej., get_item o delete_item).

3. IMPLEMENTACIÓN DEL CÓDIGO (BACKEND PYTHON)

Esta sección describe la estructura modular del *backend*, enfocada en la calidad del código, el tipado estático y la aplicación de patrones de diseño para lograr la Separación de Intereses (Separation of Concerns).

3.1. Modelo de Datos y Validación (Módulo book.py)

Se utiliza la librería Pydantic para definir el modelo de datos Book. Pydantic permite la declaración de modelos utilizando *type hinting* de Python, lo cual es crucial para la robustez de la API.

Uso y Justificación de Pydantic

- **Tipado Estático y Autodocumentación:** Asegura que los datos de entrada y salida siempre sigan una estructura definida (ej., que book_id sea un str y status sea un bool).
- **Validación Automática:** Pydantic actúa como un validador estricto. Cuando una petición POST o PUT llega a la aplicación, los datos se pasan al modelo Book(). Si los datos son inválidos o faltan campos obligatorios, Pydantic lanza una excepción automáticamente que se traduce en un error 400 Bad Request para el cliente, mejorando la experiencia del desarrollador y la seguridad de la API.

Clave Primaria

- **Campo book_id:** Definido como la clave primaria del ítem en DynamoDB.
- **Justificación:** Se utiliza un *string* aleatorio (UUID) o un *string* definido por el cliente como identificador único. Esta elección permite una distribución uniforme de los datos en las particiones de DynamoDB, lo cual es fundamental para el escalado horizontal de la base de datos.

3.2. Capa de Persistencia (Patrón Repository y Factory)

Para garantizar que la lógica de negocio (main.py) no dependa de un tipo específico de base de datos, se implementó el Patrón Repository (Repositorio), desacoplando completamente la aplicación de DynamoDB.

Patrón Factoría Abstracta / Repository

- **Clase Abstracta Database (en db.py):** Esta clase define el contrato de la base de datos. Obliga a cualquier implementación concreta a tener los cinco métodos CRUD: create_book, get_book, get_all_books, update_book, y delete_book.
- **Implementación Concreta DynamoDBDatabase (en dynamodb_db.py):** Esta clase hereda de Database e implementa la lógica específica para DynamoDB (utilizando el SDK boto3).
- **Patrón Factory:** La lógica de la aplicación utiliza una Factoría que decide qué clase concreta instanciar (ej., DynamoDBDatabase). Esto permite que, si en el futuro se desea migrar a PostgreSQL, solo sería necesario crear una clase PostgreSQLDatabase sin modificar la aplicación principal **(main.py)**.

Mapeo de Operaciones DynamoDB

Los métodos del repositorio se mapean directamente a las primitivas de DynamoDB, garantizando una interacción eficiente:

Método del Repositorio	**Operación de DynamoDB**	**Propósito**
create_book	put_item	Inserción/sobreescritura de un ítem.
get_book	get_item	Lectura de un ítem por clave primaria.
update_book	update_item	Modificación de campos.
delete_book	delete_item	Eliminación de un ítem por clave.
get_all_books	scan()	Lectura de la tabla completa. (Necesario para el requisito GET All).

Justificación de Operación y Facturación de DynamoDB

- **Operación scan():** Se utiliza para el requisito /items (GET All). La memoria advierte que la operación scan() consume muchos recursos y es ineficiente en tablas grandes. Se propone que, en un entorno de producción, se preferiría usar Índices Secundarios Globales o implementar un patrón de Paginación para optimizar esta consulta.
- **Modo de Facturación (BillingMode: PAY_PER_REQUEST):** Esta configuración, definida en la plantilla de infraestructura YAML, permite que la tabla DynamoDB opere en modo On-Demand. Esto significa que el coste se alinea con el consumo real (pago por lectura/escritura), siendo la opción ideal para las Lambdas de la arquitectura Serverless, ya que no hay coste fijo por capacidad.

3.3. Lógica de Aplicación (Handlers)

La forma en que se expone la lógica es el único punto de divergencia entre las dos arquitecturas.

Arquitectura Acoplada (ECS/Flask - main.py)

El archivo main.py utiliza el *framework* Flask para configurar las cinco rutas HTTP. Una vez que se valida el *payload* con Pydantic, la lógica llama directamente al repositorio.

- **Ejemplo de Ruta (Pseudocódigo):** La ruta @app.route('/items', methods=['POST']) recibe el *payload*, valida el Book, y llama a database_instance.create_book(validated_book).

Arquitectura Desacoplada (AWS Lambda - Handlers)

En esta arquitectura, la lógica se descompone, y cada función Lambda recibe el evento de API Gateway, lo que simplifica su código al máximo:

- **Separación de Responsabilidades:** Cada *handler* (ej., handler_create_book) solo se encarga de una única tarea: extraer los datos del evento, llamar al método específico del repositorio (ej., create_book), y devolver el resultado en el formato JSON esperado por API Gateway.
- **Eficiencia:** No hay *framework* de rutas (como Flask) ejecutándose. La función Lambda es ligera y se ejecuta solo el tiempo necesario, optimizando el consumo de GB-segundos.

4. DESPLIEGUE Y OPERACIÓN CON CLOUDFORMATION (YAML)

El despliegue de la infraestructura se realiza mediante AWS CloudFormation, una herramienta de Infraestructura como Código (IaC). Esto permite definir la arquitectura completa (red, base de datos, *load balancers*, contenedores y funciones *serverless*) en archivos YAML reutilizables.

El uso de IaC garantiza que el entorno se construya de forma repetible y predecible, eliminando la necesidad de configurar los servicios manualmente a través de la consola de AWS.

Template	Componentes Desplegados	Arquitectura Impactada
db_dynamodb.yml	Creación de la tabla DynamoDB (AWS::DynamoDB::Table).	Distintos nombres para la Acoplada y para la Desacoplada
ecr.yml	Repositorio de Contenedores ECR (AWS::ECR::Repository) y políticas de retención.	Distintos nombres para la Acoplada y para la Desacoplada
main.yml	Componentes de red, VPC Link, API Gateway principal y sus recursos asociados.	Distintos para la Acoplada y para la Desacoplada

4.1. Estructura de la Plantilla (Templates)

Para mantener la claridad y modularidad, el proyecto utiliza múltiples *templates* anidados, donde el archivo principal (main.yml) invoca a los demás.

4.3. Arquitectura Acoplada (ECS) - Despliegue del Contenedor

El *template* main.yml(P4-Aula-main books acoplada) se centra en la definición del clúster ECS y los componentes de red que permiten su acceso. Los pasos clave definidos en el YAML son:

1. **Definición de Tarea Fargate (AWS::ECS::TaskDefinition):** Se especifican los recursos de cómputo, el puerto de escucha y la ruta a la imagen Docker almacenada en ECR.
2. **Servicio ECS (AWS::ECS::Service):** Este recurso es el encargado de mantener el contenedor en ejecución de forma continua (24/7). Se le indica que lance y reemplace tareas si estas fallan.
3. **Network Load Balancer (NLB) y Target Group:** El NLB se configura para recibir tráfico en el puerto 80. Se utiliza un *Target Group* para monitorizar la salud de las tareas de Fargate y distribuir el tráfico de manera eficiente.
4. **VPC Link (AWS::ApiGateway::VpcLink):** Este es el componente crucial de la integración. Crea un enlace privado dentro de la VPC, permitiendo que el API Gateway envíe tráfico al NLB sin necesidad de exponer el *load balancer* a la red pública de internet. Esta configuración garantiza la seguridad perimetral.

4.4. Arquitectura Desacoplada (Lambda) - Despliegue Serverless

La infraestructura *Serverless* se define completamente en el *template* main.yml(P4-Aula-main books lambda), el cual explota la simplicidad de la arquitectura Desacoplada, evitando la definición de NLB y VPC Link.

1. **Funciones Lambda (AWS::Lambda::Function):** Se definen cinco recursos Lambda, uno para cada operación CRUD. En la configuración se especifica el *runtime* (Python), la cantidad de memoria (ej., 128 MB) y el *handler* (el punto de entrada de la función).
2. **Rol IAM de Ejecución:** Cada función está vinculada a un rol que le otorga los permisos mínimos necesarios. Por ejemplo, el delete_item_handler solo tiene permisos para ejecutar la acción dynamodb:DeleteItem sobre la tabla específica, adhiriéndose al principio de mínimos privilegios.
3. **API Gateway (REST API):** El *template* define la API REST y las cinco rutas HTTP (/items y /items/{id}). La integración se realiza de forma directa (AWS::ApiGateway::Integration), donde API Gateway invoca inmediatamente a la función Lambda correspondiente sin necesidad de intermediarios de red.
4. **Deployment y Stage:** Finalmente, se definen los recursos de *Deployment* y *Stage* para exponer la API bajo una URL pública y protegerla con la API Key.

5. PRUEBAS, VALIDACIÓN Y COSTES

5.1. Verificación del Funcionamiento (Postman)

Esta sección valida la correcta operación de los cinco *endpoints* CRUD en ambas arquitecturas. La clave de esta validación es la entrega de la Colección Postman y los dos Archivos de Entorno asociados, que permiten alternar instantáneamente entre la URL pública de la arquitectura Acoplada (ECS) y la Desacoplada (Lambda).

Diseño de las Pruebas

Las pruebas se enfocan en demostrar el ciclo de vida completo del dato:

1. **POST (Creación):** Verifica la respuesta HTTP 201 Created y la correcta inserción del ítem en DynamoDB.
2. **GET (by ID) y PUT:** Utiliza el ID generado en la prueba POST para asegurar que la actualización (PUT) y la lectura específica funcionen.
3. **DELETE:** Asegura que el registro puede ser eliminado de forma limpia.
4. **GET All:** Valida que se recupera la lista de ítems, verificando la operación Scan de DynamoDB.

Conclusión de la Prueba

La Colección Postman demuestra que ambas arquitecturas son funcionalmente idénticas, confirmando el éxito de la estrategia de separación de intereses al compartir la misma lógica de negocio (Patrón Repository).

5.2. Análisis de Costes (Pricing)

El análisis de costes es fundamental para justificar la elección de la arquitectura en un entorno real. Se utilizará un modelo hipotético de 100.000 peticiones de API al mes con un tamaño promedio de respuesta de 5KB y 100 milisegundos de tiempo de ejecución para Lambda y un tamaño de almacenamiento de datos de medio(0.5) GB.

Modelo de Coste Elegido

El modelo de facturación DynamoDB PAY_PER_REQUEST es crucial en esta comparativa: al ser un modelo de pago por uso, el coste de la base de datos es similar en ambas arquitecturas, concentrando la diferencia en los servicios de cómputo (ECS vs. Lambda).

Justificación DynamoDB: Con solo 100.000 operaciones de lectura/escritura al mes, el uso de la base de datos se encuentra dentro del Free Tier de AWS (25 millones de solicitudes), por lo que el coste de DynamoDB es cercano a 0,00 USD para ambas arquitecturas.

5.2.1 Arquitectura Acoplada (ECS)

Esta arquitectura tiene un coste **FIJO** por estar corriendo 24 horas al día, 7 días a la semana (24/7), incluso si no recibe tráfico.

Servicio	Mensual	Anual	Notas
DynamoDB	Almacenamiento de datos: **0.13 USD** Escritura: **0.06 USD** Lectura: **0.01 USD**	Almacenamiento de datos: **1.56 USD** Escritura: **0.72 USD** Lectura: **0.12 USD**	Para el cálculo de dividir las 100000 peticiones en escrituras y en lecturas se un escenario típico cuando se trata de libros. 20% de escrituras y 80 % de lecturas.
API Gateway	API REST: **0.35 USD**	API REST: **4.2 USD**	
ECS Fargate	ECS Fargate: **8.89 USD**	ECS Fargate: **106.68 USD**	1 tarea durante 30 días al mes
Network Load Balancer (NLB)	NLB: **16.43 USD**	**197.16 USD**	Medio GB al mes
VPC Link	VPC Link: **43.81 USD**	VPC Link: **525.72 USD**	
ECR (Repositorio)	ECR: **0.10 USD**	ECR: **0.10 USD**	
Total	**69.78 USD**	**837.36 USD**	

5.2.2 Arquitectura Desacoplada (AWS LAMBDA)

Esta arquitectura tiene un coste **VARIABLE** por petición. El coste es insignificante en inactividad.

Servicio	Mensual	Anual	Notas
DynamoDB	Almacenamiento de datos: **0.13 USD** Escritura: **0.06 USD** Lectura: **0.01 USD**	Almacenamiento de datos: **1.56 USD** Escritura: **0.72 USD** Lectura: **0.12 USD**	Para el cálculo de dividir las 100000 peticiones en escrituras y en lecturas se un escenario típico cuando se trata de libros. 20% de escrituras y 80 % de lecturas.
AWS Lambda	AWS Lambda: **0.00 USD**	AWS Lambda: **0.00 USD**	El Free Tier de Lambda (1M de peticiones y 400.000 GB-segundos) cubrirá la mayor parte del coste.
API Gateway	API HTTP: **0.10 USD**	API REST: **1.2 USD**	
Total	**0.3 USD**	**3.6 USD**	

Conclusión Arquitectónica

La implementación ha demostrado que la Arquitectura Desacoplada (Lambda) ofrece una superioridad clara para el contexto de una API CRUD moderna y con tráfico predecible:

- **Escalabilidad:** Se adapta de 0 a millones de peticiones sin intervención manual.
- **TCO (Costo Total de Propiedad):** Es radicalmente más económica para volúmenes de tráfico bajos y moderados, ya que aprovecha al máximo el modelo de Pago por Uso y el Free Tier de AWS.

La Arquitectura Acoplada (ECS Fargate), si bien es robusta y más rápida en el manejo de tráfico constante, solo se justifica en entornos con una carga base muy alta (donde el coste fijo se amortiza) o cuando se requiere *control absoluto* sobre el servidor de la aplicación (ej., mantener sesiones o cachés internos de larga duración).

La Arquitectura Acoplada (ECS Fargate), si bien es robusta y más rápida en el manejo de tráfico constante, solo se justifica en entornos con una carga base muy alta o cuando se requiere *control absoluto* sobre el servidor de la aplicación (ej., mantener sesiones o cachés internos de larga duración).

6. Uso de Herramientas de Inteligencia Artificial (IA) en el Proyecto

La Inteligencia Artificial (IA) se empleó como una herramienta de apoyo en varios puntos clave del proyecto:

- **Resolución de Errores Técnicos:** Fue utilizada como un asistente de *debugging* para identificar y solucionar **numerosos errores** que surgían al intentar desplegar los *templates* de infraestructura en **CloudFormation (YAML)**.
- **Esquematización de la Memoria:** Se empleó para ayudar a **decidir el orden** en el que se explicaban las cosas, asegurando una estructura lógica y coherente de la memoria técnica.
- **Clarificación Teórica:** Se utilizó para **entender ciertos puntos teóricos** complejos relacionados con la arquitectura *Cloud*, los servicios de AWS, y la justificación de los patrones de diseño aplicados en el código.