FaaS y Serverless

Santiago Yáñez Barajas

Nicolás David Rincón

Nicolás Quintana Cuartas

María Paula Cardona

PRESENTADO A:

Andrés Armando Sánchez Martín

ARQUITECTURA DE SOFTWARE

PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTÁ D.C

2024

CONTENIDO

URL TAG repositorios git públicos del Taller

https://github.com/Rinconjr/serverless-faas-aws-crud

Introducción

En el ámbito del desarrollo de software y la computación en la nube, la innovación ha llevado a la creación de paradigmas que simplifican y optimizan los procesos. Entre estos avances destacan la computación sin servidor (*serverless*) y las funciones como servicio (*Function as a Service* o FaaS). Estas tecnologías emergentes han revolucionado la manera en que los desarrolladores abordan la implementación de soluciones, permitiéndoles enfocarse en la lógica del negocio y delegar la gestión de la infraestructura a los proveedores de servicios en la nube.

Por estas razones exploraremos en detalle el impacto, características y aplicaciones de *serverless* y FaaS, contextualizando su relevancia dentro de la arquitectura de software moderna. Se analizarán sus principios fundamentales, ventajas y desventajas, así como su compatibilidad con conceptos de diseño como los principios SOLID. A través de ejemplos prácticos y casos de éxito, se demostrará cómo estas tecnologías están transformando sectores diversos, desde el comercio electrónico hasta el análisis de datos y la inteligencia artificial.

Definición

La computación sin servidor (*serverless*) y las funciones como servicio (*Function as a Service* o FaaS) son paradigmas que permiten a los desarrolladores enfocarse en el código sin preocuparse por la gestión de la infraestructura subyacente.

Computación sin servidor (*serverless*): Este modelo permite ejecutar aplicaciones sin la necesidad de administrar servidores. Los proveedores de servicios en la nube se encargan de la infraestructura, escalando automáticamente según la demanda y cobrando solo por el uso real de los recursos. Esto resulta en una mayor eficiencia y reducción de costos, ya que se eliminan los gastos asociados al mantenimiento de servidores inactivos.

Funciones como servicio (*FaaS*): FaaS es una categoría dentro de la computación sin servidor que permite a los desarrolladores implementar funciones individuales que se ejecutan en respuesta a eventos específicos. Estas funciones son autónomas y no requieren una infraestructura completa para operar. Los proveedores de FaaS gestionan la ejecución,

seguridad y mantenimiento de estas funciones, permitiendo a los desarrolladores concentrarse en la lógica de negocio.

Pero la realidad es que *serverless* y FaaS están relacionados, no son sinónimos. *Serverless* es un enfoque más amplio que abarca cualquier servicio donde la gestión del servidor es invisible para el usuario, incluyendo bases de datos y almacenamiento. Por otro lado, FaaS se centra específicamente en la ejecución de funciones individuales en respuesta a eventos, dentro del ecosistema *serverless*.

En la siguiente imagen es explica acerca del ecosistema Serverless:

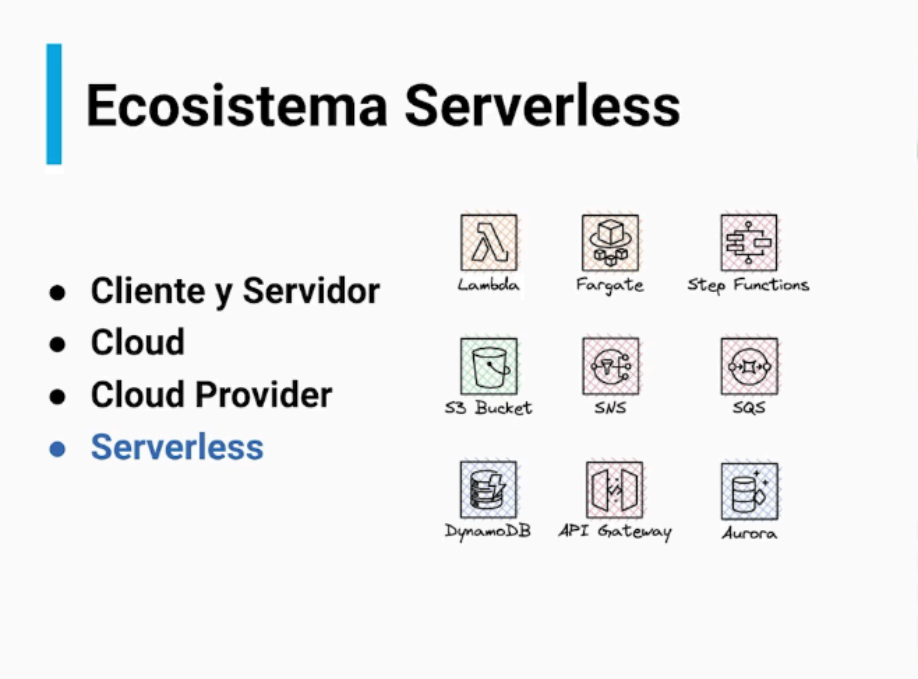


Imagen 1: Ecosistema Serverless. Fuente: Elaboración propia.

El servicio Lambda es el corazón del ecosistema serverless y representa el modelo de funciones como servicio (*Function as a Service* o FaaS). Este servicio permite ejecutar pequeñas funciones de código en respuesta a eventos como cambios en datos, solicitudes HTTP o eventos generados por otros servicios. Es altamente escalable y los usuarios solo pagan por el tiempo de ejecución real de las funciones, eliminando la necesidad de gestionar servidores o infraestructura.

Por otro lado, Fargate se centra en la ejecución de contenedores sin la necesidad de administrar servidores subyacentes. Es ideal para aplicaciones empaquetadas en contenedores y simplifica la implementación de arquitecturas basadas en microservicios al manejar automáticamente el escalado y la asignación de recursos.

Las Step Functions son un servicio de orquestación que permite coordinar múltiples servicios y tareas en flujos de trabajo definidos. Este servicio es particularmente útil para automatizar procesos empresariales o flujos complejos, permitiendo que diferentes servicios como Lambda, DynamoDB o S3 trabajen en conjunto de manera estructurada y eficiente.

S3 Bucket es el servicio de almacenamiento de objetos más utilizado dentro del ecosistema serverless. Es escalable y seguro, y permite almacenar grandes cantidades de datos no estructurados como imágenes, videos o documentos. Además, puede integrarse fácilmente con otros servicios, disparando eventos para ejecutar funciones Lambda o coordinándose con Step Functions para manejar flujos de datos.

SNS (Simple Notification Service) juega un papel clave en la mensajería en tiempo real. Este servicio permite enviar notificaciones a múltiples canales, como correos electrónicos, mensajes de texto o colas de mensajes. Es ideal para enviar alertas o notificaciones push a usuarios y para facilitar la comunicación entre diferentes servicios en una arquitectura serverless.

En este ecosistema también está SQS (Simple Queue Service), que permite la comunicación asincrónica entre diferentes componentes de una aplicación. Las colas de mensajes aseguran que los datos se entreguen de manera confiable, permitiendo desacoplar microservicios y mejorar la escalabilidad y resiliencia del sistema.

DynamoDB es una base de datos NoSQL completamente gestionada, diseñada para aplicaciones que requieren alta disponibilidad, baja latencia y escalabilidad automática. Es comúnmente utilizada para almacenar datos dinámicos como perfiles de usuario, registros de actividad o catálogos de productos.

API Gateway facilita la interacción entre clientes (como aplicaciones web o móviles) y servicios backend. Este servicio permite crear y administrar APIs REST o WebSocket, manejando tareas como autenticación, limitación de solicitudes y transformación de datos, actuando como un puente eficiente en las arquitecturas serverless.

Por último, Aurora es una base de datos relacional de alto rendimiento diseñada específicamente para la nube. Compatible con MySQL y PostgreSQL, ofrece escalabilidad

automática y alta disponibilidad, siendo ideal para aplicaciones que requieren bases de datos transaccionales con capacidades avanzadas.

En conjunto, estos servicios forman el ecosistema serverless, que permite construir aplicaciones modernas, escalables y altamente eficientes, liberando a los desarrolladores de la gestión de infraestructura y permitiéndoles centrarse en la lógica y funcionalidad de sus aplicaciones.

Características

En cuanto a las características, comenzaremos analizando FaaS (Function as a Service), destacando las más relevantes que lo definen y lo convierten en una solución innovadora dentro del ámbito de la computación en la nube.

- **Ejecución basada en eventos**: Las funciones se activan en respuesta a eventos específicos, como cambios en una base de datos, solicitudes HTTP, eventos IoT o tareas programadas.
- **Sin necesidad de gestionar servidores**: Los desarrolladores no se ocupan de la administración, mantenimiento o configuración de servidores; el proveedor del servicio maneja toda la infraestructura subyacente.
- **Escalabilidad dinámica y automática**: Las funciones se ajustan automáticamente para manejar la carga de trabajo según la demanda, aumentando o reduciendo los recursos utilizados.
- **Modelo de computación granular**: El código se organiza en pequeñas unidades funcionales que ejecutan tareas específicas, proporcionando una estructura modular.
- **Modelo de precios basado en uso real**: Los costos están directamente relacionados con el número de invocaciones y el tiempo de ejecución de las funciones, en lugar de una tarifa fija o basada en infraestructura.
- **Integración con otros servicios en la nube**: Las funciones se integran de manera nativa con servicios del mismo ecosistema del proveedor, como almacenamiento, bases de datos o colas de mensajes.
- **Inicio rápido (cold start y warm start)**: Las funciones pueden iniciarse casi instantáneamente, aunque pueden variar en velocidad dependiendo si están en estado "cold" (apagadas) o "warm" (activas).
- **Aislamiento de funciones**: Cada función se ejecuta de manera independiente, lo que asegura que fallos en una no afecten a las demás.

- **Control basado en configuración**: Las funciones pueden configurarse para ejecutarse en ciertos intervalos, frente a eventos concretos o según reglas específicas definidas por el usuario.
- **Manejo de sesiones efímeras**: FaaS está optimizado para manejar estados temporales, ya que las funciones no retienen datos entre ejecuciones. Si se requiere almacenamiento, debe utilizarse un servicio externo.
- **Modelo reactivo**: La ejecución ocurre en tiempo real en respuesta a eventos, lo que permite una alta capacidad de respuesta en sistemas dinámicos.
- **Sin infraestructura fija**: No se asignan recursos dedicados de manera permanente a las funciones; los recursos se asignan temporalmente según la necesidad.

El modelo Serverless se ha convertido en una opción atractiva para el desarrollo de aplicaciones modernas gracias a una serie de características clave que lo distinguen dentro de la computación en la nube. A continuación, se detallan sus principales caracteristicas:

- **Delegación de la administración de servidores**
  Serverless elimina la necesidad de gestionar la infraestructura subyacente. Los tiempos de ejecución de programas o servicios se definen y manejan automáticamente, sin requerir supervisión constante ni la instalación de plugins. El software opera de manera completamente autónoma, liberando a los desarrolladores de tareas operativas.
- **Escalado automático**
  Este modelo permite ajustar automáticamente los recursos de rendimiento y memoria según las necesidades de la aplicación. Cuando se requiere mayor capacidad para manejar cargas de trabajo más pesadas, el proceso de escalado se ejecuta de forma dinámica y transparente, asegurando un desempeño óptimo.
- **Automatización y tolerancia a errores**
  Serverless incorpora mecanismos de tolerancia a fallos en las aplicaciones, reduciendo la necesidad de configurar protocolos de contingencia. Esto significa que las aplicaciones están preparadas para manejar errores de manera automática, garantizando una mayor estabilidad y continuidad en los servicios.
- **Ejecución basada en eventos**
  Serverless permite a los desarrolladores ejecutar código en respuesta a eventos específicos, como solicitudes HTTP, cambios en bases de datos, mensajes de colas de tareas o eventos programados. Este enfoque reactivo reduce el uso de recursos y permite arquitecturas más eficientes.
- **Bases de datos y almacenamiento Serverless**
  Aprovecha servicios en la nube diseñados para ser escalables y completamente gestionados. Esto incluye soluciones de bases de datos y almacenamiento serverless,

que permiten almacenar y recuperar datos de forma eficiente sin necesidad de gestión adicional por parte del usuario.

- **Naturaleza sin estado**
  Las funciones serverless se ejecutan como tareas independientes y no mantienen estado entre invocaciones. Esto permite un alto grado de escalabilidad, ya que los recursos pueden ser asignados y reutilizados de manera eficiente por el proveedor de nube.
- **Escalado y asignación dinámica de recursos**
  Serverless asegura que los recursos solo se asignen cuando una función está activa, evitando costos innecesarios y optimizando la utilización de infraestructura. Este modelo de asignación basado en uso es ideal para cargas de trabajo variables o eventos intermitentes.

Historia y evolución

La historia del Function as a Service (FaaS) muestra su transformación de una idea emergente en una pieza clave de la computación en la nube moderna, a continuación se detalla cómo fue el paso en la historia del FaaS hasta el dia de hoy:

1. **Orígenes y primeros desarrollos:**

Antes de la popularización del término "FaaS", ya existían servicios que abstraían la gestión de servidores. Por ejemplo, en 2010, PiCloud ofrecía una plataforma de computación en la nube que permitía a los usuarios ejecutar funciones sin preocuparse por la infraestructura, sentando las bases para lo que hoy conocemos como FaaS.

2. **Introducción de AWS Lambda:**

En noviembre de 2014, Amazon Web Services (AWS) lanzó AWS Lambda, considerado el primer servicio FaaS de un proveedor de nube de gran escala. Lambda permitió a los desarrolladores ejecutar código en respuesta a eventos específicos sin necesidad de gestionar servidores, marcando un hito en la computación sin servidor.

3. **Adopción por otros proveedores de nube:**

Tras el éxito de AWS Lambda, otros gigantes de la nube introdujeron sus propias soluciones FaaS:

- **Google Cloud Functions**: Lanzado en 2017, permitió a los desarrolladores ejecutar funciones en respuesta a eventos en la infraestructura de Google Cloud.

- **Microsoft Azure Functions**: Introducido en 2016, ofreció una plataforma para ejecutar funciones desencadenadas por eventos en el ecosistema de Azure.
- **IBM Cloud Functions**: Basado en el proyecto de código abierto Apache OpenWhisk, proporcionó una solución FaaS en la nube de IBM.

4. **Expansión y adopción:**

Con el tiempo, FaaS se integró en arquitecturas de microservicios y aplicaciones basadas en eventos, permitiendo a las empresas desarrollar aplicaciones más escalables y eficientes. La comunidad de código abierto también contribuyó con proyectos como Apache OpenWhisk, que ofreció una alternativa FaaS de código abierto.

5. **Estado actual:**

Hoy en día, FaaS es una parte integral de las estrategias de computación en la nube, facilitando el desarrollo de aplicaciones sin la carga de gestionar infraestructura. Su adopción continúa creciendo, impulsada por la necesidad de soluciones más ágiles y escalables en el desarrollo de software.

Ahora, el concepto de Serverless ha pasado de ser un enfoque innovador a convertirse en un estándar en la computación en la nube. Desde sus primeros pasos con Google App Engine en 2006 hasta su consolidación con AWS Lambda, el modelo serverless ha transformado la manera en que se desarrollan, despliegan y escalan las aplicaciones modernas.

Serverless

1. **Primeros indicios y conceptos iniciales:**

Antes de la aparición formal del término "serverless", ya existían servicios que abstraían la gestión de servidores. En 2006, Google lanzó Google App Engine, una plataforma que permitía a los desarrolladores crear y alojar aplicaciones web sin preocuparse por la infraestructura subyacente. Este servicio ofrecía un entorno escalable y gestionado, sentando las bases para lo que posteriormente se conocería como computación sin servidor.

2. **Aparición del término "Serverless":**

El término "serverless" comenzó a ganar popularidad alrededor de 2012, cuando se utilizó para describir arquitecturas que no requerían la gestión explícita de servidores por parte de los desarrolladores. Este enfoque se centraba en permitir que las aplicaciones se ejecutaran en entornos completamente gestionados por proveedores de servicios en la nube, eliminando la necesidad de administrar la infraestructura.

3. **Lanzamiento de AWS Lambda y consolidación del modelo:**

Un hito significativo en la historia de Serverless fue el lanzamiento de AWS Lambda por Amazon Web Services en noviembre de 2014. AWS Lambda permitió a los desarrolladores ejecutar código en respuesta a eventos específicos sin necesidad de gestionar servidores, popularizando el modelo de "Function as a Service" (FaaS) y consolidando el paradigma serverless en la industria.

4. **Adopción por otros proveedores de nube:**

Tras el éxito de AWS Lambda, otros proveedores de servicios en la nube introdujeron sus propias soluciones serverless:

- **Google Cloud Functions**: Lanzado en 2017, ofreció una plataforma para ejecutar funciones en respuesta a eventos dentro del ecosistema de Google Cloud.
- **Microsoft Azure Functions**: Introducido en 2016, proporcionó una solución similar en la plataforma Azure, permitiendo a los desarrolladores crear aplicaciones basadas en eventos sin gestionar servidores.
- **IBM Cloud Functions**: Basado en el proyecto de código abierto Apache OpenWhisk, ofreció una plataforma serverless en la nube de IBM.

5. **Expansión y diversificación del ecosistema serverless:**

Con el tiempo, el ecosistema serverless se expandió más allá de FaaS, incorporando servicios como bases de datos, almacenamiento y orquestación sin servidor. Esto permitió a los desarrolladores construir aplicaciones completas utilizando componentes serverless, beneficiándose de escalabilidad automática, modelos de pago por uso y una reducción significativa en la complejidad operativa.

6. **Estado actual y perspectivas futuras:**

Hoy en día, la computación sin servidor es una parte integral de las estrategias de desarrollo en la nube, facilitando la creación de aplicaciones más ágiles y escalables. La adopción de arquitecturas serverless continúa creciendo, impulsada por la necesidad de soluciones más eficientes y flexibles en el desarrollo de software.

Ventajas y desventajas

Function as a Service (FaaS) y **Serverless** son enfoques modernos de computación en la nube diseñados para simplificar el desarrollo de aplicaciones al eliminar la necesidad de administrar

servidores de forma directa. Aunque los términos están relacionados, FaaS es un subconjunto del paradigma Serverless.

1. **FaaS**: Es un modelo de computación basado en funciones. Los desarrolladores escriben pequeñas funciones que se activan en respuesta a eventos específicos. Por ejemplo, una solicitud HTTP puede activar una función en AWS Lambda, y esta ejecuta su lógica, escala automáticamente y luego se apaga. Ideal para tareas event-driven o basadas en procesos aislados.
2. **Serverless**: Es un concepto más amplio que incluye FaaS, bases de datos Serverless, almacenamiento, y otros servicios que no requieren gestionar servidores. Aquí, el proveedor en la nube (AWS, Azure, Google Cloud) maneja la infraestructura completa, mientras tú te enfocas en la lógica del negocio.

Ventajas

FaaS

1. **Escalabilidad automática**: Cuando una función es llamada muchas veces (por ejemplo, cientos de usuarios accediendo simultáneamente), el sistema genera instancias adicionales automáticamente. Esto elimina la preocupación por configurar manualmente servidores adicionales.
2. **Costos optimizados**: A diferencia de los servidores tradicionales que cobran por hora o mes, FaaS cobra por cada milisegundo de ejecución, reduciendo costos significativamente si tu aplicación no está en uso constante.
3. **Desarrollo centrado en funciones pequeñas**: Con FaaS, puedes dividir una aplicación en funciones individuales. Esto promueve la modularidad y facilita la reutilización de código.

Serverless

1. **Sin gestión de servidores**: No necesitas actualizar sistemas operativos, instalar parches de seguridad ni preocuparte por el hardware. El proveedor maneja todo eso por ti.
2. **Ecosistema completo**: Serverless combina múltiples servicios (almacenamiento, bases de datos, mensajería, etc.), todos escalables y gestionados automáticamente, lo que te permite crear aplicaciones complejas sin preocuparte por la infraestructura.

Desventajas

FaaS

1. **Cold starts**: Si una función no se usa durante cierto tiempo, se "apaga". Al volver a activarse, toma tiempo inicializar el entorno, lo que puede afectar el rendimiento de aplicaciones críticas en tiempo real.
2. **Restricciones técnicas**: Los proveedores de FaaS limitan el tiempo de ejecución (por ejemplo, AWS Lambda tiene un máximo de 15 minutos por invocación), memoria y tamaño de las funciones.

Serverless

1. **Bloqueo de proveedor**: Usar servicios Serverless suele implicar configuraciones propietarias (por ejemplo, AWS DynamoDB o Google Firestore). Migrar aplicaciones de un proveedor a otro puede requerir reescribir partes del sistema.
2. **Costos impredecibles**: Aunque el modelo de pago por uso parece económico, un pico inesperado en la actividad del sistema puede generar facturas altas, especialmente en sistemas mal configurados o con errores.

Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

1. Procesamiento de Eventos en Tiempo Real

- **Problema**: Necesitas procesar datos en tiempo real, como análisis de logs, detección de fraudes o procesamiento de transmisiones de video/audio.
- **Uso**: Usar FaaS (como AWS Lambda) para procesar eventos generados por dispositivos IoT, aplicaciones móviles o servicios web. Por ejemplo:
    - Filtrar y transformar datos enviados desde sensores IoT.
    - Analizar los logs en tiempo real para detectar patrones anómalos.

2. Automatización de Tareas Programadas

- **Problema**: Ejecutar tareas periódicas como respaldos de bases de datos, generación de reportes o limpieza de datos antiguos.
- **Uso**: Configurar funciones FaaS para activarse en horarios específicos mediante un servicio como AWS CloudWatch o Azure Timer. Ejemplo:
    - Automatizar la generación y envío de reportes diarios.
    - Borrar automáticamente registros de bases de datos más antiguos de cierta fecha.

3. Aplicaciones Web y Backends ligeros

- **Problema**: Construir una aplicación web o móvil con demandas impredecibles de tráfico.
- **Uso**: Implementar el backend en Serverless con API Gateway y FaaS:
    - API Gateway para recibir solicitudes HTTP/HTTPS.
    - Funciones FaaS para manejar lógica de negocio, como autenticación, CRUD o procesamiento de pagos.

Ejemplo: Un e-commerce donde cada solicitud para consultar productos, agregar al carrito o procesar pagos activa una función FaaS.

4. Procesamiento de Archivos

- **Problema**: Manejar operaciones como la conversión, compresión o análisis de archivos cargados por usuarios.
- **Uso**: Configurar FaaS para que se activen cuando un usuario suba un archivo a un sistema de almacenamiento (como Amazon S3):
    - Procesar imágenes (redimensionarlas, cambiar formato).
    - Analizar archivos de texto para extraer datos o ejecutar validaciones.

Ejemplo: Una aplicación de fotos que ajusta automáticamente el tamaño de las imágenes al cargarlas.

5. Chatbots y Asistentes Virtuales

- **Problema**: Crear un chatbot que responda a solicitudes en tiempo real.
- **Uso**: FaaS para manejar las interacciones del usuario, conectándose a APIs de lenguaje natural como Google Dialogflow o AWS Lex:
    - Responder a preguntas frecuentes.
    - Realizar tareas específicas como consultar un saldo o reservar un servicio.

6. Análisis de Datos y Machine Learning

- **Problema**: Procesar grandes volúmenes de datos o realizar predicciones basadas en modelos entrenados.
- **Uso**: Combinar FaaS con almacenamiento y herramientas de análisis:
    - Activar funciones para analizar datos en lotes pequeños desde un sistema de almacenamiento como S3 o Google Cloud Storage.
    - Utilizar funciones para inferencia en tiempo real de modelos pre-entrenados.

Ejemplo: Una función que predice la demanda de productos basada en datos históricos cada vez que se carga un nuevo dataset.

7. Gestión de Notificaciones

- **Problema**: Enviar notificaciones personalizadas a usuarios en respuesta a eventos.
- **Uso**: FaaS para enviar correos, SMS o notificaciones push al activarse ciertos triggers:
    - Notificar al cliente cuando su pedido esté listo.
    - Enviar alertas de seguridad en sistemas financieros al detectar movimientos sospechosos.

8. Sistemas de Cola y Procesamiento por Lotes

- **Problema**: Procesar tareas que llegan en lotes, como solicitudes de usuarios o mensajes en un sistema de mensajería.
- **Uso**: FaaS para procesar mensajes en una cola (como Amazon SQS o Google Pub/Sub):
    - Procesar órdenes de compra una a una desde una cola.
    - Consumir mensajes de una aplicación distribuida para mantener la sincronización.

9. Aplicaciones Temporales

- **Problema**: Requerir infraestructura para un evento o campaña específica con alta carga por un tiempo limitado.
- **Uso**: Usar Serverless para manejar aplicaciones efímeras, como encuestas masivas o eventos en vivo:
    - Una encuesta en línea que recibe miles de respuestas durante un evento.
    - Un sistema de votaciones en tiempo real.

10. Modernización de Sistemas Legados

- **Problema**: Migrar aplicaciones monolíticas a una arquitectura más escalable y moderna.
- **Uso**: Dividir el monolito en funciones independientes mediante FaaS:
    - Migrar tareas específicas, como generación de reportes o validaciones, a funciones Serverless.

Casos de aplicación (Ejemplos y casos de éxito en la industria)

El modelo serverless ha transformado la manera en que las empresas desarrollan y despliegan aplicaciones. A continuación, se presentan algunos ejemplos de casos de éxito y aplicaciones en la industria:

1. Streaming y Procesamiento en Tiempo Real

Ejemplo: Netflix

- **Aplicación:** Netflix utiliza serverless para procesar eventos en tiempo real y ajustar las recomendaciones personalizadas. Usando servicios como AWS Lambda, Netflix monitorea millones de eventos diarios para mejorar la experiencia del usuario.
- **Impacto:** Se reducen costos de infraestructura, y se logra un escalado dinámico durante los picos de tráfico, como estrenos de series o películas populares.

2. E-Commerce y Manejo de Inventarios

Ejemplo: Zalando

- **Aplicación:** Zalando, un gigante del comercio electrónico, emplea arquitecturas serverless para manejar eventos relacionados con el inventario y procesar cambios en tiempo real. Esto incluye notificaciones automáticas a los usuarios cuando un producto vuelve a estar disponible.
- **Impacto:** Mejor gestión de inventarios, reducción en el tiempo de procesamiento y mayor satisfacción del cliente.

3. Internet de las Cosas (IoT)

Ejemplo: Philips Hue

- **Aplicación:** Philips utiliza una arquitectura serverless para conectar y administrar millones de dispositivos IoT (bombillas inteligentes). AWS Lambda facilita la comunicación entre dispositivos y servidores en la nube, gestionando funciones como encender luces, ajustar la intensidad o programar horarios.
- **Impacto:** Operación eficiente con baja latencia, soporte para millones de usuarios globalmente y escalabilidad sin complejidad adicional.

4. Análisis de Datos y Machine Learning

Ejemplo: Coca-Cola

- **Aplicación:** Coca-Cola implementó un sistema basado en serverless para el análisis de datos de máquinas expendedoras inteligentes. Utilizando Google Cloud Functions, recopilan y procesan datos como transacciones y mantenimiento.
- **Impacto:** Reducción de costos de operación y análisis en tiempo real para mejorar la toma de decisiones.

5. Automatización de Flujos de Trabajo

Ejemplo: iRobot (Roomba)

- **Aplicación:** iRobot utiliza servicios serverless para gestionar las actualizaciones de software y el monitoreo de los robots aspiradores en tiempo real. Los datos enviados por los dispositivos se procesan con Lambda y se almacenan en DynamoDB.
- **Impacto:** Actualizaciones más rápidas, gestión de grandes volúmenes de datos y mejor experiencia para los usuarios finales.

6. Marketing Digital y Personalización de Contenidos

Ejemplo: The New York Times

- **Aplicación:** En eventos como elecciones presidenciales, The New York Times utiliza arquitecturas serverless para escalar rápidamente aplicaciones que presentan resultados en tiempo real a millones de usuarios.
- **Impacto:** Reducción significativa de costos y capacidad de manejar picos de tráfico sin interrupciones.

7. Aplicaciones Financieras

Ejemplo: Capital One

- **Aplicación:** Capital One utiliza arquitecturas serverless para procesar transacciones, detectar fraudes en tiempo real y generar reportes financieros automatizados.
- **Impacto:** Mayor rapidez en los tiempos de respuesta y una infraestructura optimizada para picos de demanda.

8. Creación y Procesamiento de Contenido Multimedia

Ejemplo: BBC

- **Aplicación:** BBC usa serverless para convertir contenido multimedia a formatos compatibles con múltiples dispositivos. Usando AWS Lambda, automatizan la codificación de videos y la distribución a través de plataformas digitales.
- **Impacto:** Mejora en tiempos de entrega y reducción de costos al no mantener infraestructura dedicada.

¿Qué tan común es Serverless hoy en día?

Serverless es muy común en la actualidad y se ha consolidado como una solución estándar en arquitecturas modernas debido a sus ventajas como la escalabilidad automática, el modelo de pago por uso y la simplificación del desarrollo al delegar la gestión de infraestructura al proveedor. Grandes plataformas como AWS Lambda, Google Cloud Functions y Azure Functions lideran su adopción, siendo utilizadas en casos como APIs backend, procesamiento en tiempo real y automatización de flujos de trabajo. Aunque enfrenta desafíos como latencias iniciales y costos altos en usos constantes, su popularidad sigue creciendo por la flexibilidad y eficiencia que ofrece.

Análisis de Principios SOLID vs Serverless

	Serverless	**Faas**
Single Responsibility	Alto	Alto
Open/Closed	Alto	Medio
Liskov Substitution	Medio	Bajo
Interface Segregation	Alto	Alto
Dependency Inversion	Medio	Bajo

1.

Single Responsibility

- **Serverless:** Se basa en la separación de responsabilidades al dividir las aplicaciones en microservicios independientes, cada uno con una función clara. Esto facilita el cumplimiento del principio.

- **FaaS:** Similarmente, FaaS permite que cada función se desarrolle con una única responsabilidad específica. Sin embargo, la fragmentación puede complicar la coordinación entre funciones cuando aumenta la escala.

2. Open/Closed

- **Serverless:** Este principio se cumple al permitir la extensión del sistema mediante la adición de nuevos servicios sin necesidad de modificar los existentes. Serverless está diseñado para crecer horizontalmente con facilidad.
- **FaaS:** Aunque también permite agregar nuevas funciones sin modificar las ya existentes, la fragmentación puede dificultar la gestión de extensiones en sistemas más grandes, lo que limita parcialmente su capacidad para cumplir este principio.

3. Liskov Substitution

- **Serverless:** Puede cumplir este principio si los servicios están diseñados para ser intercambiables sin afectar la lógica de negocio. Sin embargo, esto depende del diseño específico del sistema, por lo que su cumplimiento es moderado.
- **FaaS:** Tiende a tener mayor dificultad para cumplir este principio debido a la dependencia explícita entre funciones. Sustituir funciones puede ser complejo, ya que suelen estar altamente acopladas a las interacciones específicas del sistema.

4. Interface Segregation

- **Serverless:** Permite diseñar interfaces específicas y evitar la sobrecarga, ya que cada servicio puede tener sus propias responsabilidades bien delimitadas.
- **FaaS:** Similar a Serverless, cumple bien este principio porque cada función puede implementar interfaces pequeñas y específicas, evitando la complejidad.

5. Dependency Inversion

- **Serverless:** Puede cumplir parcialmente este principio, siempre y cuando las dependencias entre servicios se gestionen adecuadamente mediante inyección de dependencias o abstracciones bien diseñadas. Sin embargo, no siempre es fácil de implementar en sistemas distribuidos.
- **FaaS:** Presenta mayores desafíos en este aspecto, ya que las dependencias entre funciones suelen ser directas y difíciles de abstraer. Esto complica el cumplimiento del principio en sistemas complejos.

Análisis de Atributos de Calidad vs Serverless

	Serverless	Faas
Idoneidad funcional	Alto	Alto
Eficiencia	Medio	Alto
Compatibilidad	Medio	Medio
Usabilidad	Alto	Medio
Confiabilidad	Medio	Alto
Seguridad	Alto	Alto
Mantenibilidad	Alto	Medio
Portabilidad	Medio	Bajo

Idoneidad Funcional

- **Serverless:** Muestra un desempeño alto, ya que permite gestionar aplicaciones completas y escalables, facilitando su implementación en proyectos de gran alcance. Esto lo hace ideal para sistemas que requieren modularidad y alta disponibilidad.
- **FaaS:** También tiene un desempeño alto, diseñado para la ejecución eficiente de tareas aisladas. Es especialmente útil en casos donde se requiere una respuesta rápida para funciones específicas.

Eficiencia

- **Serverless:** Aunque eficiente en aplicaciones amplias, puede enfrentar problemas de latencia inicial (*cold starts*), lo que afecta su rendimiento en sistemas que necesitan respuestas inmediatas.
- **FaaS:** Sobresale en eficiencia, ya que está optimizado para tiempos de ejecución rápidos en funciones específicas, lo que lo hace más adecuado para tareas pequeñas y recurrentes.

Compatibilidad

- **Serverless:** Muestra niveles medios de compatibilidad debido a su dependencia de los proveedores de servicios en la nube. Esto puede limitar la integración con otras tecnologías o servicios externos en entornos multinube.

- **FaaS:** Al igual que Serverless, enfrenta desafíos similares en compatibilidad, ya que depende de la infraestructura específica del proveedor, complicando la interoperabilidad.

Usabilidad

- **Serverless:** Ofrece una experiencia amigable para los desarrolladores, con herramientas que simplifican la gestión de aplicaciones completas. Es especialmente ventajoso en sistemas más complejos.
- **FaaS:** También tiene alta usabilidad, aunque puede requerir configuraciones adicionales y fragmentar el flujo de trabajo, especialmente en arquitecturas con muchas funciones interrelacionadas.

Confiabilidad

- **Serverless:** Destaca por su capacidad para manejar sistemas complejos de manera resiliente, garantizando la estabilidad de las aplicaciones a pesar de las altas cargas.
- **FaaS:** Es confiable en funciones individuales, pero en sistemas con dependencias interconectadas puede enfrentar desafíos que impacten su desempeño general.

Seguridad

- **Serverless:** Ofrece altos niveles de seguridad mediante herramientas avanzadas para gestionar aplicaciones completas. Su enfoque amplio requiere configuraciones cuidadosas para minimizar riesgos.
- **FaaS:** Tiene una ligera ventaja en seguridad, ya que las funciones están aisladas, reduciendo el impacto de posibles vulnerabilidades en el sistema.

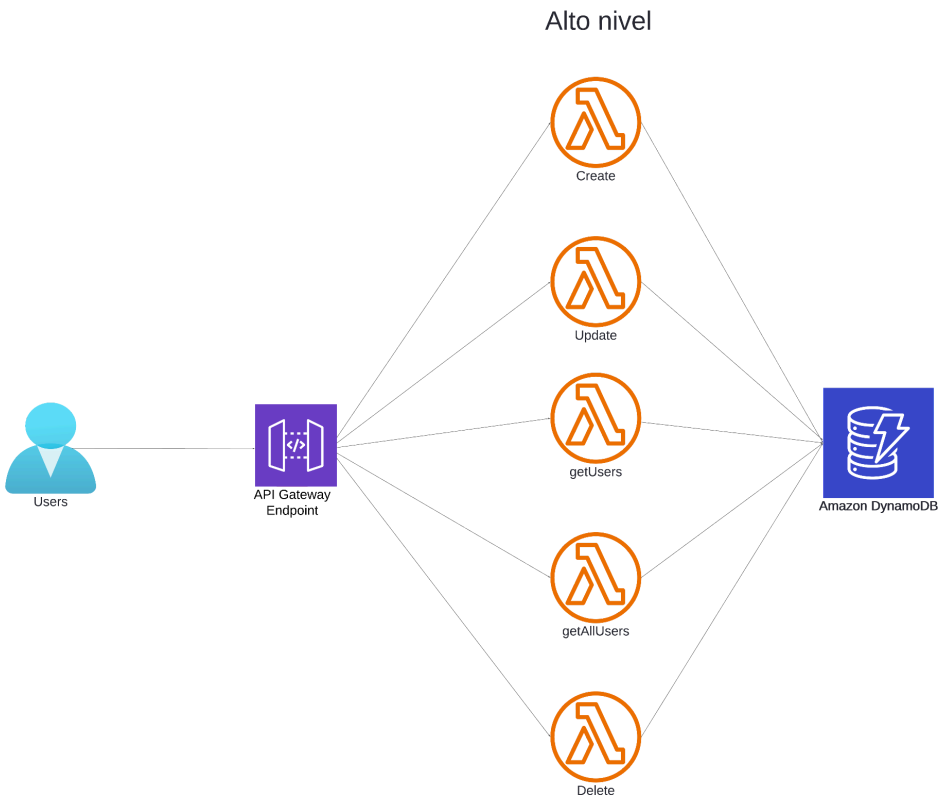
Mantenibilidad

- **Serverless:** Sobresale en este aspecto, ya que simplifica la gestión de aplicaciones al eliminar la necesidad de infraestructura física. Esto reduce el esfuerzo de mantenimiento, especialmente en sistemas amplios.
- **FaaS:** Aunque funcional, puede incrementar la carga de mantenimiento debido a la fragmentación de funciones y las dependencias entre ellas, dificultando su gestión en arquitecturas grandes.

Portabilidad

- **Serverless:** Obtiene una calificación superior, ya que tiene mayor capacidad para adaptarse a diferentes proveedores de servicios en la nube. Esto facilita su migración entre plataformas con menor esfuerzo.
- **FaaS:** Es más limitado en portabilidad, ya que está estrechamente vinculado a la infraestructura del proveedor, lo que complica la migración a otros entornos sin realizar ajustes significativos.

Arquitectura de alto nivel



Este sistema serverless basado en AWS permite manejar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) relacionadas con usuarios de manera eficiente y escalable. Los usuarios interactúan con el sistema a través de un API Gateway, que actúa como punto de entrada principal. Este componente recibe las solicitudes HTTP realizadas por los usuarios y las redirige hacia las funciones Lambda correspondientes.

Las Funciones Lambda están diseñadas para ejecutar una operación específica del sistema:

- Create para crear nuevos usuarios.

- Update para actualizar información existente.
- getUsers para obtener información de un usuario específico.
- getAllUsers para recuperar la lista completa de usuarios.
- Delete para eliminar usuarios del sistema.

Cada función Lambda realiza la lógica necesaria para procesar la solicitud y, si es requerido, interactúa directamente con Amazon DynamoDB, una base de datos NoSQL que almacena toda la información de los usuarios. DynamoDB garantiza operaciones de lectura y escritura rápidas y confiables, soportando la alta demanda del sistema.

El flujo general del sistema sigue estos pasos:

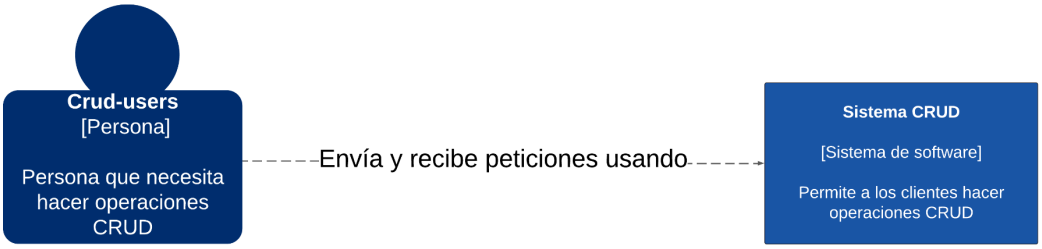
- El usuario envía una solicitud HTTP al sistema.
- El API Gateway recibe la solicitud y la redirige a la función Lambda apropiada.
- La función Lambda procesa la solicitud e interactúa con DynamoDB si se requiere manipular datos.
- La función Lambda genera una respuesta, que el API Gateway devuelve al usuario.

Este diseño serverless es ideal para aplicaciones modernas, ofreciendo flexibilidad, escalabilidad automática y costos operativos reducidos, al tiempo que garantiza una experiencia fluida y confiable para los usuarios.

C4Model

Diagrama de conexto

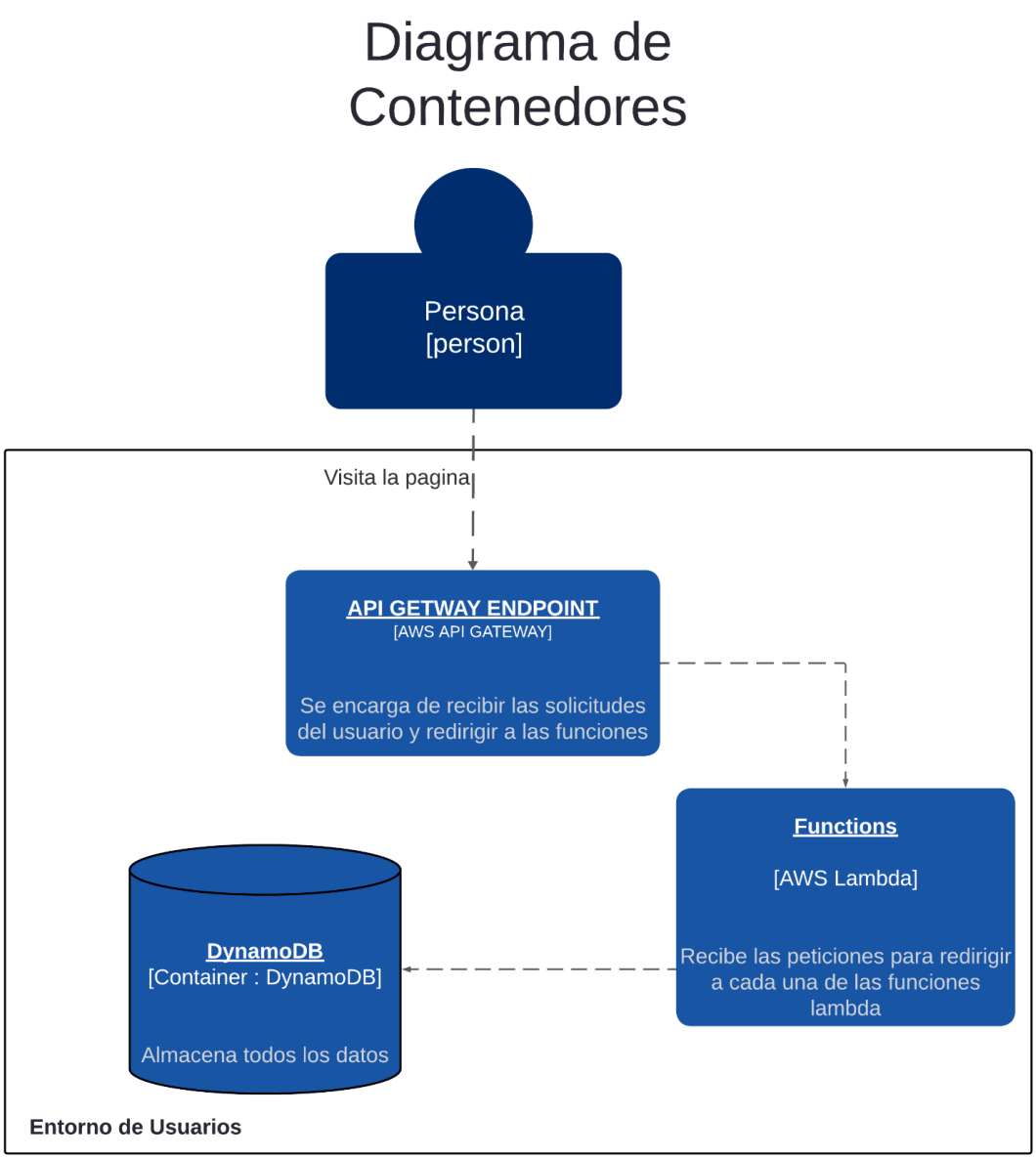
Diagrama de Contexto



Este diagrama de contexto representa la interacción entre un usuario (denominado "Crud-users") y un sistema serverless para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar). A continuación, se describe cada elemento del diagrama:

1. **Crud-users (Usuario)**:
    - Representa a una persona que necesita realizar operaciones CRUD. Este usuario envía solicitudes y recibe respuestas desde el sistema.
    - Su papel es consumir las funcionalidades que el sistema proporciona, como la gestión de datos u operaciones específicas.
2. **Sistema CRUD (Sistema serverless)**:
    - Es un sistema de software basado en una arquitectura serverless que utiliza funciones como servicio (FaaS, Functions as a Service) para permitir que los usuarios realicen operaciones CRUD.
    - El sistema responde dinámicamente a las solicitudes del usuario, ejecutando funciones específicas para cada operación solicitada (por ejemplo, almacenamiento, consultas o modificaciones en una base de datos).
3. **Interacción entre el usuario y el sistema**:
    - El usuario interactúa con el sistema a través de peticiones (por ejemplo, usando API REST o eventos). El sistema procesa estas peticiones usando funciones serverless alojadas en una plataforma como AWS Lambda, Azure Functions, o Google Cloud Functions.
    - La comunicación es bidireccional: el usuario envía datos o solicitudes al sistema, y el sistema devuelve los resultados de las operaciones.

Diagrama de contenedores



Este diagrama de contenedores ilustra la arquitectura de un sistema serverless que utiliza servicios de AWS, específicamente diseñado para manejar peticiones de usuarios y almacenar datos de manera eficiente. A continuación, se describe cada elemento:

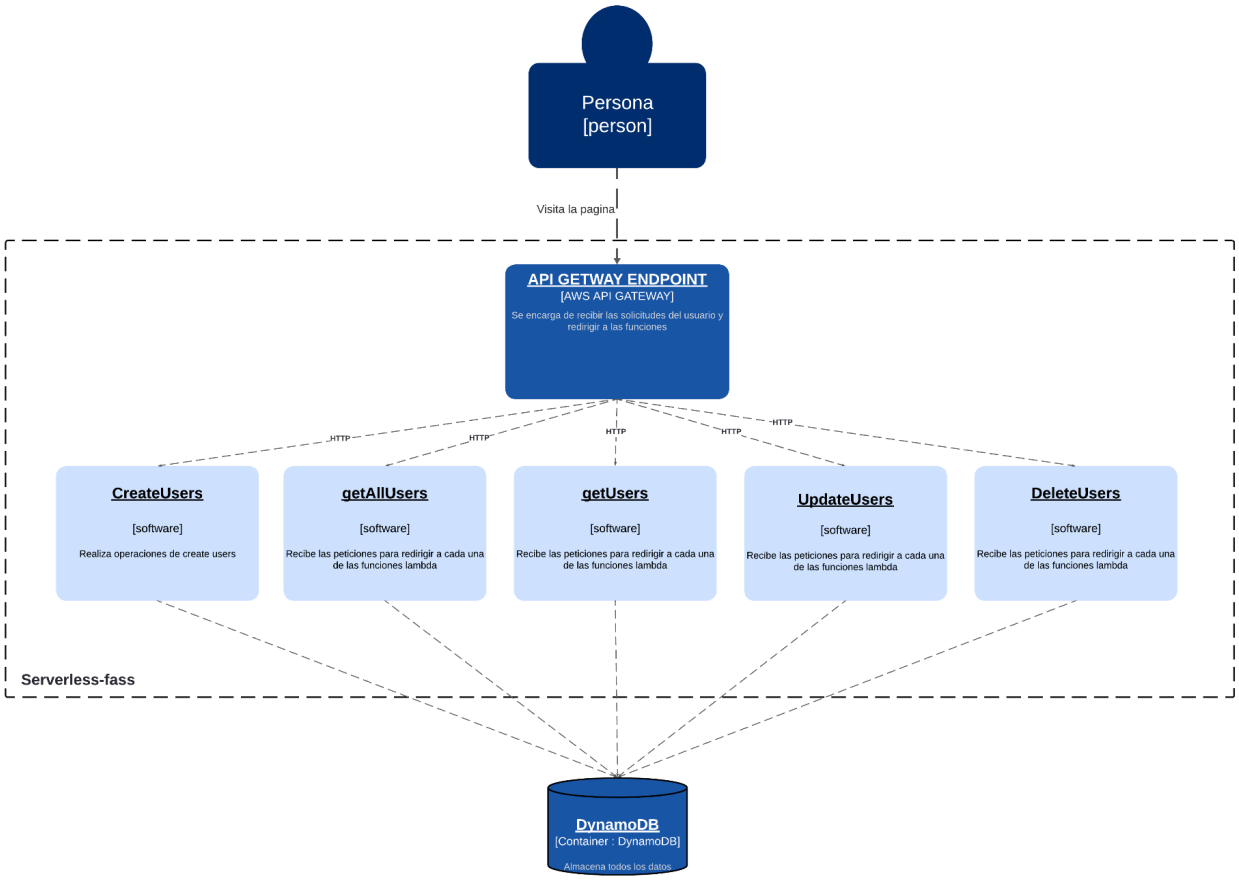
1. **Persona [person]**:
    - Representa al usuario final que interactúa con el sistema. Este usuario accede a la página web o aplicación asociada al sistema para realizar operaciones.
2. **API Gateway Endpoint [AWS API Gateway]**:

- Es el punto de entrada principal del sistema. Se encarga de recibir las solicitudes de los usuarios (por ejemplo, a través de métodos HTTP como GET, POST, PUT, DELETE).
- Este servicio actúa como un intermediario que redirige las solicitudes a las funciones Lambda correspondientes, dependiendo de la lógica de negocio implementada.

3. **Functions [AWS Lambda]**:
    - Representa las funciones serverless que ejecutan la lógica de negocio. Cada función está diseñada para manejar tareas específicas basadas en las solicitudes recibidas del API Gateway.
    - Este componente es altamente escalable y ejecuta código en respuesta a eventos, eliminando la necesidad de administrar servidores.
4. **DynamoDB [Contenedor: DynamoDB]**:
    - Es la base de datos NoSQL utilizada para almacenar todos los datos relacionados con el sistema.
    - Las funciones Lambda interactúan con DynamoDB para leer, escribir, actualizar o eliminar datos según sea necesario, asegurando un almacenamiento eficiente y de baja latencia.
5. **Flujo del sistema**:
    - **Paso 1**: El usuario visita la página web o aplicación, realizando una acción que genera una solicitud.
    - **Paso 2**: La solicitud llega al **API Gateway Endpoint**, que decide a qué función Lambda dirigirla.
    - **Paso 3**: La **Function** correspondiente se activa y ejecuta la lógica de negocio necesaria.
    - **Paso 4**: Si la operación requiere interacción con los datos, la función Lambda se comunica con **DynamoDB** para procesar la información.
    - **Paso 5**: La función Lambda devuelve la respuesta al API Gateway, que a su vez la entrega al usuario

Diagrama de componentes

Digrama Componentes



Este diagrama representa una arquitectura serverless basada en AWS, diseñada para gestionar usuarios mediante operaciones CRUD (Crear, Leer, Actualizar y Eliminar). A continuación, se describe cada componente y su interacción en el sistema:

- **Persona [person]:** El usuario final que interactúa con el sistema a través de una página web o aplicación, enviando solicitudes relacionadas con la gestión de usuarios, como agregar, consultar, actualizar o eliminar información.
- **API Gateway Endpoint [AWS API Gateway]:** Es el punto de entrada del sistema, responsable de recibir las solicitudes del usuario y redirigirlas a las funciones Lambda específicas según la operación solicitada. Proporciona una capa de comunicación eficiente y segura.
- **Funciones Lambda (CreateUsers, getAllUsers, getUsers, UpdateUsers, DeleteUsers):** Representan la lógica de negocio del sistema, donde cada función maneja una operación específica:

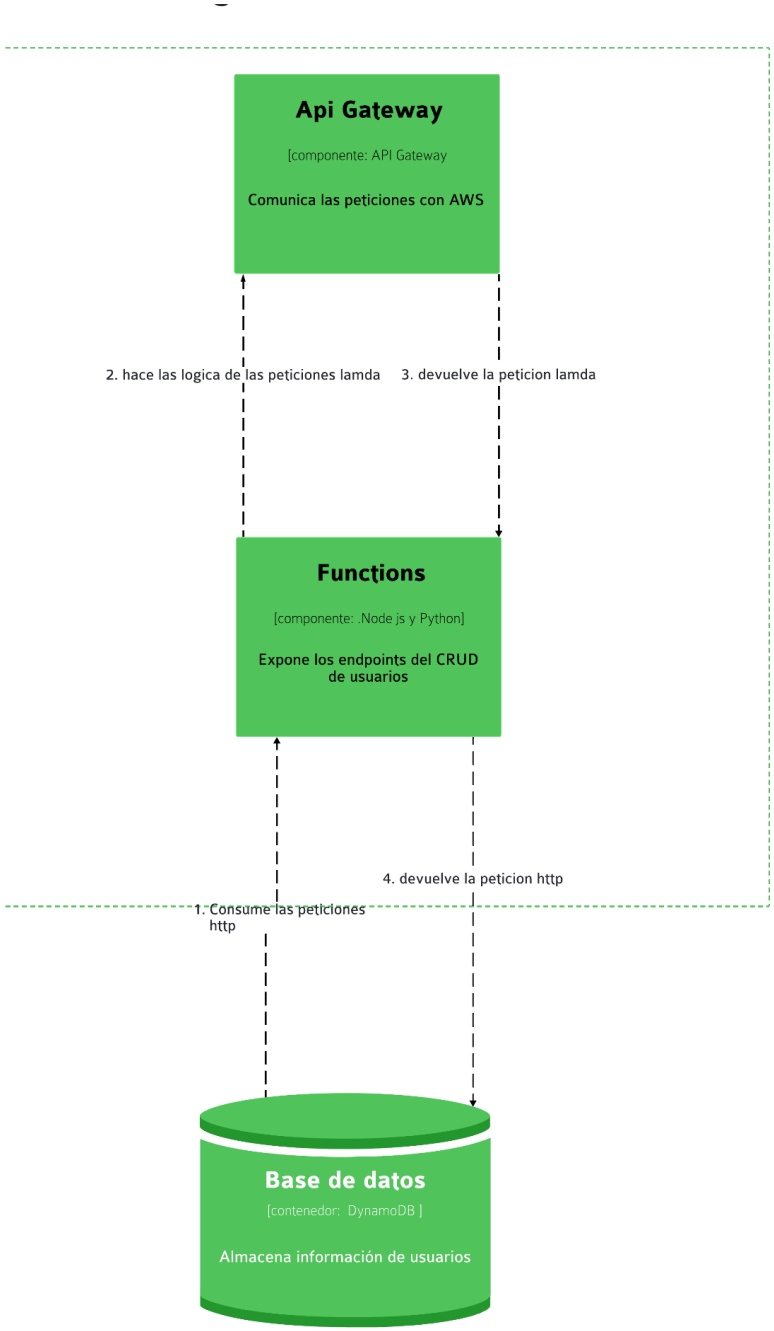
    - **CreateUsers:** Procesa la creación de nuevos usuarios.
    - **getAllUsers:** Recupera todos los usuarios almacenados.
    - **getUsers:** Obtiene información específica de un usuario.
    - **UpdateUsers:** Actualiza datos de usuarios existentes.
    - **DeleteUsers:** Elimina usuarios del sistema. Estas funciones son ejecutadas de manera independiente y escalan automáticamente según la demanda.
- DynamoDB [Contenedor: DynamoDB]: Es la base de datos NoSQL donde se almacenan los datos de los usuarios. DynamoDB interactúa con las funciones Lambda para realizar operaciones como almacenar, consultar, actualizar o eliminar registros.

Flujo del Sistema:

- El usuario accede al sistema y realiza una solicitud (como agregar o consultar un usuario).
- El API Gateway recibe la solicitud y la redirige a la función Lambda correspondiente.
- La función Lambda procesa la solicitud según la lógica definida y, si es necesario, interactúa con DynamoDB para gestionar los datos.
- La función Lambda envía la respuesta al API Gateway, que se encarga de devolverla al usuario.

Este diseño aprovecha los beneficios de la arquitectura serverless, como escalabilidad, costos reducidos y mantenimiento simplificado, mientras asegura una interacción eficiente y segura entre los componentes del sistema.

Diagrama Dynamic C4



El diagrama representa una arquitectura serverless implementada con servicios de AWS para manejar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) relacionadas con usuarios. El flujo del sistema y sus componentes clave se detallan a continuación:

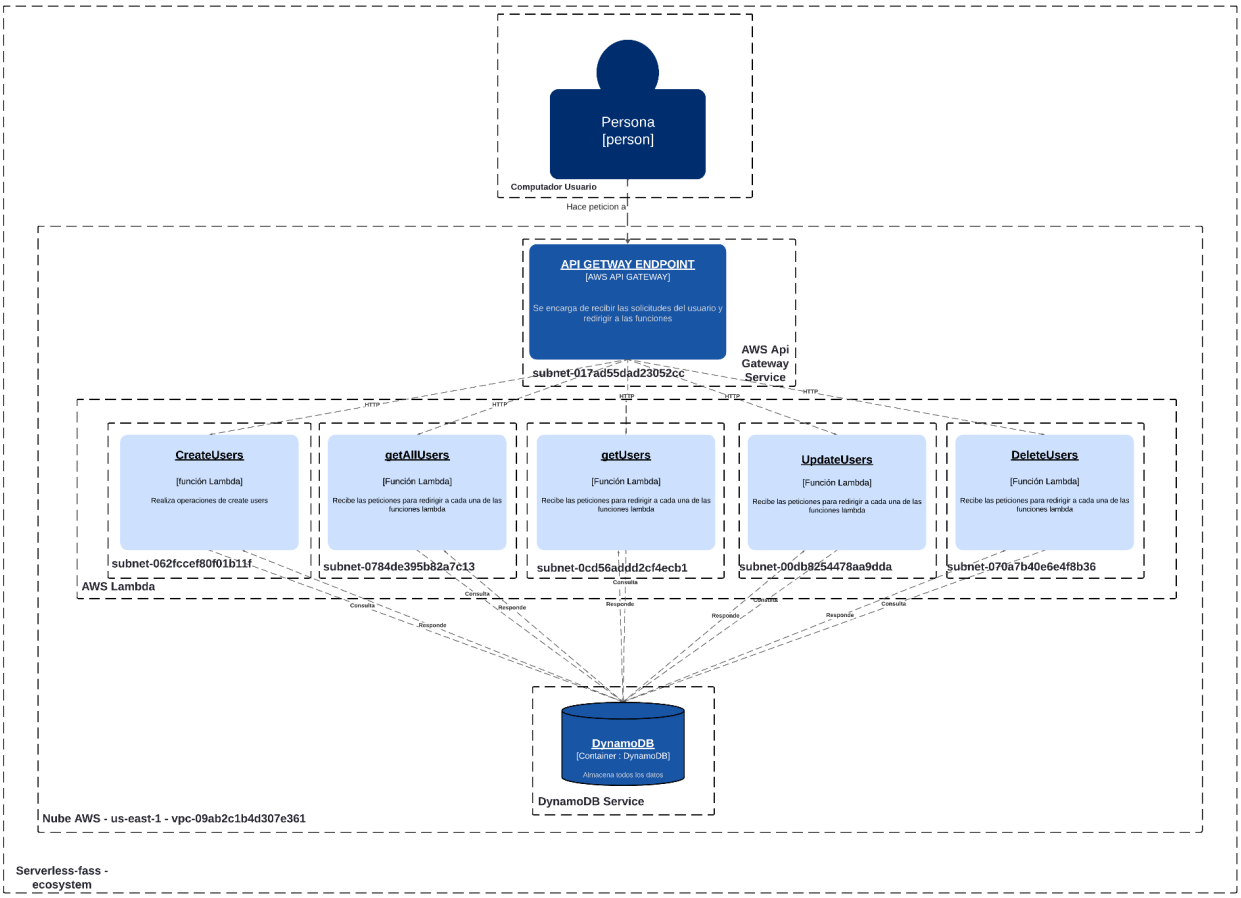
- **API Gateway**: Actúa como el punto de entrada principal, encargándose de recibir las solicitudes HTTP de los usuarios. Este componente redirige dichas solicitudes a las funciones Lambda específicas y, tras procesarlas, devuelve la respuesta al cliente. Es responsable de gestionar la comunicación entre el usuario y las capas internas del sistema.
- **Funciones Lambda (Functions)**: Implementadas en Node.js y Python, estas funciones ejecutan la lógica de negocio. Cada función está diseñada para manejar una operación CRUD específica. Las peticiones del cliente, una vez redirigidas por el API Gateway, son procesadas por estas funciones, que pueden incluir validaciones, transformaciones y cálculos. Además, las funciones Lambda interactúan directamente con la base de datos para gestionar los datos según sea necesario.
- **Base de Datos (DynamoDB)**: Es una base de datos NoSQL que almacena la información de los usuarios. DynamoDB proporciona almacenamiento rápido y confiable, permitiendo que las funciones Lambda realicen operaciones de lectura, escritura, actualización o eliminación de datos de manera eficiente.

Flujo de Operaciones:

1. El usuario realiza una solicitud HTTP, como crear o consultar información de un usuario.
2. El **API Gateway** recibe la solicitud y la redirige a la función Lambda correspondiente.
3. La función Lambda procesa la solicitud, aplicando la lógica necesaria, y si es requerido, interactúa con **DynamoDB** para manipular los datos.
4. Una vez completada la operación, la función Lambda envía la respuesta al **API Gateway**, que se encarga de devolverla al usuario.

Este diseño combina escalabilidad, eficiencia y simplicidad, aprovechando las ventajas de una arquitectura serverless para ofrecer una solución robusta y de bajo costo para la gestión de usuarios.

Diagrama Despliegue C4



El sistema representa un ecosistema serverless en AWS diseñado para manejar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) de usuarios. Los principales componentes son:

- **Persona [person]**: El usuario final interactúa con el sistema a través de un navegador o aplicación cliente, enviando solicitudes como crear, consultar, actualizar o eliminar usuarios.
- **API Gateway Endpoint**: Es el punto de entrada al sistema. Recibe las solicitudes del usuario y las redirige a las funciones Lambda específicas según la operación requerida, asegurando una gestión eficiente y segura de cada solicitud.
- **Funciones Lambda (CreateUsers, getAllUsers, getUsers, UpdateUsers, DeleteUsers)**: Cada función maneja una operación CRUD particular:
    - **CreateUsers**: Crea nuevos registros de usuarios.

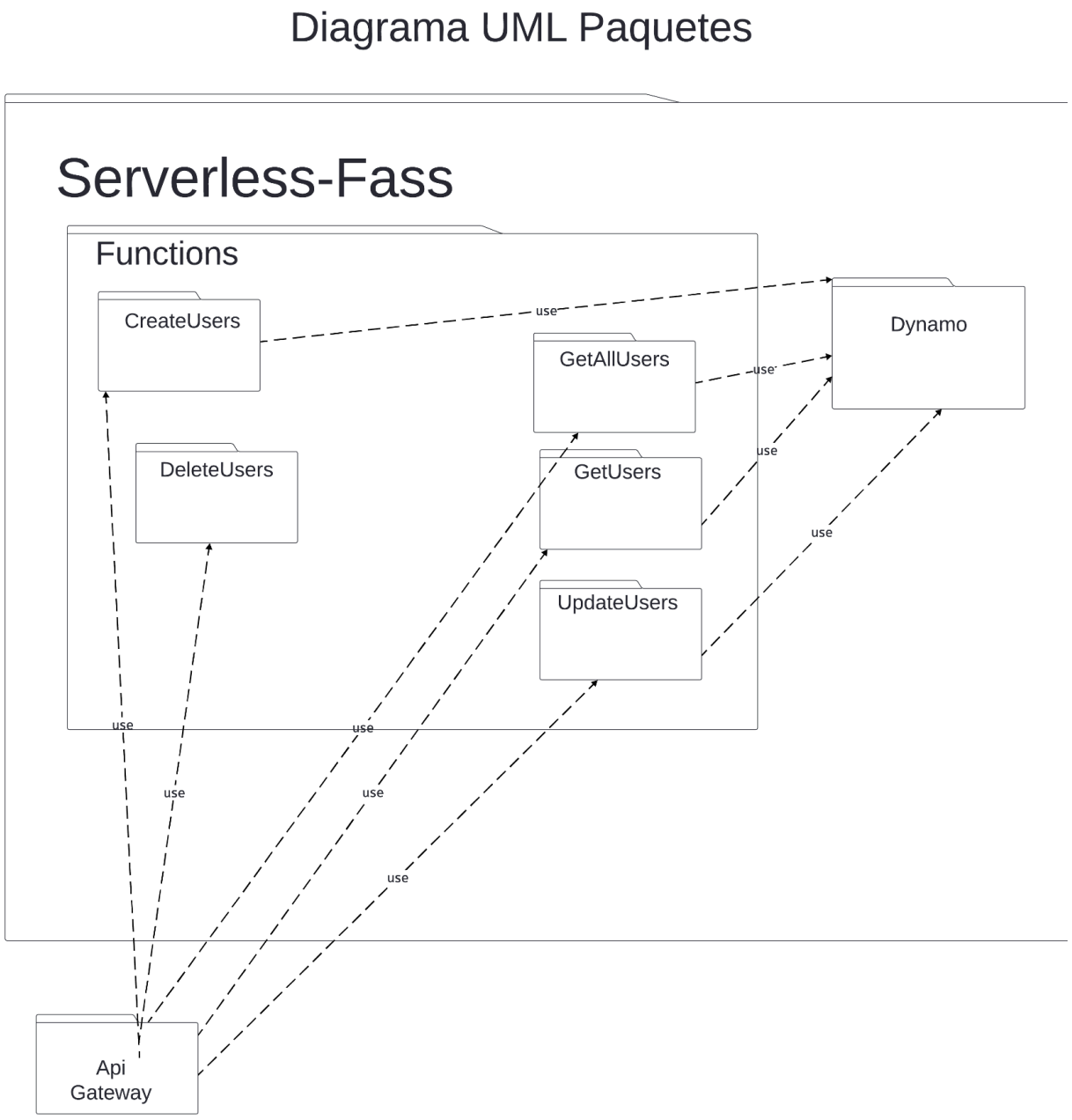
    - **getAllUsers**: Recupera la lista completa de usuarios.
    - **getUsers**: Consulta información de un usuario específico.
    - **UpdateUsers**: Actualiza los datos de un usuario.
    - **DeleteUsers**: Elimina usuarios existentes. Estas funciones se ejecutan en subredes específicas dentro de la nube de AWS, garantizando escalabilidad y control de recursos.
- **DynamoDB**: Es la base de datos NoSQL que almacena los datos de los usuarios. Proporciona almacenamiento de alta disponibilidad y bajo tiempo de respuesta, interactuando con las funciones Lambda para manejar las operaciones de datos.

Flujo de Operaciones

1. El usuario envía una solicitud desde su dispositivo.
2. El **API Gateway** recibe y redirige la solicitud a la función Lambda correspondiente.
3. La **función Lambda** ejecuta la lógica de negocio requerida e interactúa con **DynamoDB** si es necesario para gestionar datos.
4. La respuesta generada por la función Lambda es enviada al **API Gateway**, que la devuelve al usuario.

Este ecosistema utiliza los beneficios de AWS, como la escalabilidad automática, alta disponibilidad y costos optimizados. La integración entre **API Gateway**, **Funciones Lambda** y **DynamoDB** asegura un manejo eficiente y confiable de las operaciones CRUD para ofrecer una experiencia fluida a los usuarios y un sistema robusto para la gestión de datos.

Diagrama de paquetes UML de cada componente



El diagrama representa una arquitectura serverless organizada en tres paquetes principales: API Gateway, Functions y Dynamo, diseñada para manejar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) de usuarios de manera eficiente y escalable.

API Gateway: Es el punto de entrada al sistema que recibe las solicitudes de los usuarios y las redirige hacia las funciones correspondientes en el paquete Functions. Este componente asegura que cada solicitud sea procesada correctamente, actuando como intermediario entre los usuarios y la lógica del negocio.

- **Functions:** Contiene las funciones Lambda que implementan la lógica de negocio del sistema, cada una encargada de una operación específica:
- **CreateUsers:** Crea nuevos usuarios.
- **DeleteUsers:** Elimina usuarios existentes.
- **GetAllUsers:** Recupera la lista de todos los usuarios.
- **GetUsers:** Obtiene información específica de un usuario.
- **UpdateUsers:** Actualiza los datos de un usuario existente. Estas funciones reciben las solicitudes desde el API Gateway y procesan la lógica necesaria, interactuando con el paquete Dynamo para gestionar los datos según sea necesario.
- **Dynamo:** Representa la base de datos NoSQL (Amazon DynamoDB), que almacena y gestiona los datos de los usuarios. Este componente interactúa con las funciones Lambda para realizar operaciones de lectura, escritura, actualización o eliminación de datos, garantizando un almacenamiento rápido, confiable y de alta disponibilidad.

El flujo del sistema sigue estos pasos: el usuario envía una solicitud al API Gateway, que la redirige a la función Lambda adecuada. La función procesa la solicitud, interactúa con Dynamo si es necesario y devuelve la respuesta al API Gateway, que a su vez se la entrega al usuario. Esta arquitectura modular y escalable asegura una gestión eficiente de las operaciones CRUD de usuarios, aprovechando las capacidades de AWS para un rendimiento óptimo.

Muestra de funcionalidad, en vivo o video

Para la muestra de funcionalidad hemos decidido grabar el siguiente video donde se explica la estructura del taller y se muestran todos sus componentes al igual que el producto final en funcionamiento

Link del video: https://www.youtube.com/watch?v=ViLrbnWSJFQ

Conclusiones

- *Serverless* y *Function as a Service* (FaaS) simplifican el desarrollo de software al eliminar la gestión de infraestructura, permitiendo a los desarrolladores centrarse en la lógica de negocio.
- El análisis de los principios SOLID y atributos de calidad confirma que *serverless* y FaaS favorecen sistemas modulares, confiables y mantenibles.
- La adopción masiva por parte de empresas líderes y startups resalta su relevancia en la transformación digital.
- *Serverless* y FaaS son herramientas clave para organizaciones que buscan innovar, optimizar recursos y responder a las demandas dinámicas del mercado moderno.

Referencias

- Red Hat. (n.d.). *¿Qué es FaaS (Function-as-a-Service)?*. Red Hat. Recuperado de https://www.redhat.com/es/topics/cloud-native-apps/what-is-faas#:~=La%20FaaS%20permite%20que%20los,lo%20convirtiera%20a%20diferentes%20formatos.

- IONOS. (n.d.). *¿Qué es Function as a Service (FaaS)?*. IONOS Digital Guide. Recuperado de https://www.ionos.com/es-us/digitalguide/servidores/know-how/function-as-a-service-faas/.

- Azion. (n.d.). *¿Qué es Serverless?*. Azion. Recuperado de https://www.azion.com/es/learning/serverless/que-es-serverless/.

- Aplyca. (2021). *Serverless Computing o Informática sin Servidores*. Blog de Aplyca. Recuperado de https://www.aplyca.com/blog/serverless-computing-o-informatica-sin-servidores.

- IBM. (n.d.). *¿Qué es Serverless?*. IBM. Recuperado de https://www.ibm.com/es-es/topics/serverless

- Genbeta. (2019). *¿Qué es Serverless y por qué deberías adoptarlo en el desarrollo de tu próxima aplicación?*. Genbeta. Recuperado de https://www.genbeta.com/desarrollo/que-serverless-que-adoptarlo-desarrollo-tu-proxima-aplicacion

- Wikipedia. (n.d.). *Serverless computing*. Wikipedia. Recuperado de https://en.wikipedia.org/wiki/Serverless_computing
- AWS. (n.d.). *What is Serverless Computing?*. Amazon Web Services. Recuperado de https://aws.amazon.com/es/what-is/serverless-computing/
- Administración de Sistemas. (2021). *Serverless computing: La evolución de la nube*. Administración de Sistemas. Recuperado de https://administracciondesistemas.com/serverless-computing-la-evolucion-de-la-nube/
- Humble, C. (2021, marzo 29). Matthew Clark on the BBC's migration from LAMP to the cloud with AWS Lambda, React and CI/CD. InfoQ. https://www.infoq.com/podcasts/bbc-aws-lambda-react-cicd/
- Kowalcze, K. (2021, septiembre 13). Serverless use cases: see what companies follow the serverless approach. CrustLab. https://crustlab.com/blog/serverless-use-cases-see-what-companies-follow-the-serverless-approach/
- Rehemägi, T. (2020, julio 16). Serverless framework: The Coca-cola case study. Dashbird. https://dashbird.io/blog/serverless-case-study-coca-cola/
- Retter, M. (2020, julio 30). Netflix AWS: The Netflix serverless case study. Dashbird. https://dashbird.io/blog/serverless-case-study-netflix/
- Us, A. (s/f). What are serverless examples? 8 real-world use cases of serverless technology. Rumblefish.dev. Recuperado el 19 de noviembre de 2024, de https://www.rumblefish.dev/blog/post/what-are-serverless-examples-8-real-world-use-cases-of-serverless-technology/
- (S/f-a). Amazon.com. Recuperado el 19 de noviembre de 2024, de https://aws.amazon.com/solutions/case-studies/innovators/irobot/
- (S/f-b). Wsj.com. Recuperado el 19 de noviembre de 2024, de https://www.wsj.com/articles/new-york-times-cto-makes-case-for-serverless-computing-1511993275
- (S/f-c). Amazon.com. Recuperado el 19 de noviembre de 2024, de https://aws.amazon.com/es/solutions/case-studies/capital-one-all-in-on-aws/

- Casos De Uso De La Tecnología Serverless Que No Conocías. (2021, May 27). Retrieved November 19, 2024, from Codster website: https://codster.io/blog/cloud-computing/casos-de-uso-tecnologia-serverless/