**MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS**

**31105151 - TRABAJO FIN DE MÁSTER**

**ARQUITECTURAS ORIENTADAS A SERVICIOS**

# API Management en Arquitecturas Serverless

Manuel Moro Álvarez

Directora: Dra. Elena Ruiz Larrocha

Curso 2022/23 (Convocatoria Septiembre/2023)

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

31105151 - Trabajo Fin de Máster
Línea de trabajo: Arquitecturas orientadas a servicios.
Tipo B
Título: Api Management En Arquitecturas Serverless

Alumno: Manuel Moro Álvarez

Centro Asociado UNED Illes Balears, Palma, España.

manolomoro@gmail.com

Directora: Dra. Elena Ruiz Larrocha

Departamento de Ingeniería del Software y Sistemas Informáticos,
ETSI Informática UNED, Madrid, España.

elena@issi.uned.es

**Abstract.**

[es] Este trabajo aborda la transformación tecnológica de aplicaciones monolíticas hacia arquitecturas serverless mediante la modularización de aplicaciones y la gestión de APIs. Se exploran los retos inherentes a las aplicaciones monolíticas y se propone una solución basada en la adopción de arquitecturas serverless y API Management. Se ofrece una comparativa de las infraestructuras serverless frente a las tradicionales basadas en servidores, y se discuten los beneficios del manejo de APIs en un entorno serverless. Un caso práctico detallado ilustra el proceso de transformación, desde el análisis y planificación, hasta la descomposición de la aplicación, pruebas, despliegue, monitoreo, ajustes, optimización y escalado. Los resultados muestran mejoras significativas en eficiencia y flexibilidad, a la vez que se identifican desafíos en el proceso de transformación. Esta investigación sienta las bases para futuros estudios en este campo emergente.

[en] This study addresses the technological transformation of monolithic applications towards serverless architectures through application modularization and API management. The inherent challenges of monolithic applications are explored, and a solution based on the adoption of serverless architectures and API Management is proposed. A comparison of serverless infrastructures versus traditional server-based ones is provided, and the benefits of API management in a serverless environment are discussed. A detailed practical case illustrates the transformation process, from analysis and planning, through application decomposition, testing, deployment, monitoring, adjustments, optimization, and scaling. The results show significant improvements in efficiency and flexibility while identifying challenges in the transformation process. This research lays the groundwork for future studies in this emerging field.

**Keywords:**

[es] Arquitectura Serverless, Gestión de APIs, Transformación Tecnológica, Aplicaciones Monolíticas, Modularización de Aplicaciones, Optimización, Escalado, Pruebas de Software, AWS Lambda, Amazon API Gateway.

[en] Serverless Architecture, API Management, Technological Transformation, Monolithic Applications, Application Modularization, Optimization, Scaling, Software Testing, AWS Lambda, Amazon API Gateway.

# Índice de Contenido

Ilustraciones

# 1. Introducción

Históricamente, el desarrollo de sistemas y aplicaciones en entornos empresariales no siempre ha seguido una arquitectura previamente estructurada y planificada. Las áreas de negocio, responsables de la generación de valor en las compañías, han influido más que los departamentos de Tecnologías de la Información (TI), relegados a un papel secundario y sin capacidad real de decisión en los procesos de negocio que deben implementar.

La priorización de proyectos enfocados en la evolución de las capacidades del producto en lugar de proyectos de "TI para TI" ha llevado a muchas empresas a un punto de no retorno. Estas organizaciones enfrentan dificultades para evolucionar debido a la carencia de fundamentos tecnológicos sólidos sobre los que seguir construyendo. Las aplicaciones desarrolladas en este contexto suelen ser monolíticas, con interconexiones de baja calidad tecnológica, altos costes de mantenimiento y una complejidad creciente que dificulta su gestión y evolución.

Para superar este punto de bloqueo, es necesario abordar proyectos de transformación tecnológica que proporcionen una nueva base para dar continuidad al negocio. Las áreas de infraestructura y arquitectura deben liderar la propuesta tecnológica desde el departamento de TI, asegurando que sea viable, eficiente y posible de implementar sin interrumpir las operaciones actuales.

En este contexto, las arquitecturas serverless y una correcta implementación de la gestión de APIs pueden ser fundamentales para diseñar una solución adecuada. Este trabajo tiene como objetivo examinar la problemática asociada con la transformación tecnológica, considerando tanto la infraestructura como las arquitecturas serverless en entornos de nube y on-premise. Además, se enfoca en la estrategia de modularización necesaria para superar las aplicaciones monolíticas, generando un ecosistema de aplicaciones eficientemente conectadas mediante capas de APIs gestionadas por un servicio de API Management.

Para diseñar una estrategia eficaz en la transformación tecnológica, es fundamental analizar las herramientas actuales en el mercado para implementar aplicaciones basadas en arquitecturas serverless y API management. Esta revisión permitirá identificar las opciones más adecuadas

para abordar la problemática planteada y facilitar la toma de decisiones en la adopción de tecnologías apropiadas.

A modo de resumen, el principal **objetivo** de este trabajo fin de máster es **analizar la transformación tecnológica en entornos empresariales** y **proponer soluciones basadas en arquitecturas serverless y API management** para superar las limitaciones de aplicaciones monolíticas, así como **analizar un caso práctico** de transformación tecnológica de una de aplicación monolítica a una basada en microservicios, logrando una mayor eficiencia y flexibilidad en los procesos de negocio.

# 2. Análisis de la problemática y necesidades de transformación tecnológica

## a. Introducción

La problemática de las aplicaciones monolíticas en entornos empresariales surge de varios factores que afectan su escalabilidad, mantenimiento y evolución. En aplicaciones monolíticas, los componentes del sistema están muy interconectados, lo que complica su escalabilidad. Al escalar una aplicación monolítica, es necesario replicar todos los componentes, incluso si solo una parte experimenta una mayor demanda, lo que conduce a un uso ineficiente de recursos.

El mantenimiento y las actualizaciones de aplicaciones monolíticas se vuelven más difíciles debido a que combinan diversas funcionalidades en una sola estructura, lo que aumenta el riesgo de errores y extiende el tiempo de resolución de problemas. Además, suelen estar construidas con una única tecnología o lenguaje de programación, lo que limita la capacidad de adoptar nuevas tecnologías o enfoques que podrían mejorar el rendimiento y la eficiencia del sistema. La integración de nuevas funcionalidades o servicios también se ve limitada, ya que cualquier adición puede afectar múltiples componentes del sistema.

El despliegue de una aplicación monolítica puede ser un proceso lento y costoso, ya que cualquier cambio requiere desplegar toda la aplicación en

lugar de solo actualizar el componente modificado. Esto puede resultar en tiempos de inactividad prolongados y un mayor riesgo de interrupción del servicio.

Para enfrentar estos desafíos, muchas empresas están adoptando enfoques basados en arquitecturas de microservicios y serverless, que permiten dividir aplicaciones en componentes más pequeños y autónomos. Estas arquitecturas facilitan la escalabilidad, el mantenimiento y la evolución de las aplicaciones, al tiempo que permiten una mayor flexibilidad en la adopción de tecnologías y la integración de servicios.

### b. Identificación de las limitaciones de las aplicaciones monolíticas

En esta sección, se identifican y analizan las principales limitaciones de las aplicaciones monolíticas que afectan su escalabilidad, mantenimiento y evolución en entornos empresariales.

**Escalabilidad**

- Dificultad para escalar componentes individuales: En las aplicaciones monolíticas, todos los componentes de la aplicación están diseñados para trabajar juntos como una única unidad. Por lo tanto, si un componente específico necesita más recursos debido a la alta demanda, no se puede escalar de forma independiente sin escalar toda la aplicación. Esto significa que si, por ejemplo, una función de la aplicación está experimentando una demanda particularmente alta, no puedes simplemente añadir más recursos a esa función específica. En lugar de eso, tendrías que escalar toda la aplicación, lo que puede ser costoso e ineficiente.
- Uso ineficiente de recursos: Cuando escalas una aplicación monolítica, básicamente estás duplicando toda la aplicación, incluso las partes que no necesitan escalado. Esto puede llevar a un uso ineficiente de los recursos, ya que estás utilizando más capacidad de procesamiento y memoria para componentes de la aplicación que no necesitan esos recursos adicionales. Además, la replicación de toda la aplicación también puede llevar a un mayor tiempo de inactividad durante el proceso de escalado, ya que toda la aplicación debe ser replicada y desplegada.

**Mantenimiento y actualizaciones**

- Riesgo de errores: En una aplicación monolítica, todas las funcionalidades están estrechamente interrelacionadas y dependen unas de otras. Esto significa que un error en una parte de la aplicación puede tener efectos en cascada y afectar a otras partes de la aplicación. Además, debido a la complejidad de estas aplicaciones, la detección y resolución de errores puede ser un proceso complicado y laborioso. Cada cambio, por pequeño que sea, puede requerir que se pruebe toda la aplicación, lo que puede llevar mucho tiempo y aumentar la posibilidad de que se introduzcan nuevos errores.

- Limitación tecnológica: Las aplicaciones monolíticas suelen estar construidas en un único lenguaje de programación y tecnología. Esto puede limitar la capacidad de la empresa para adoptar nuevas tecnologías que podrían mejorar la eficiencia y el rendimiento de la aplicación. Además, si el lenguaje de programación o la tecnología elegidos inicialmente se vuelven obsoletos o ya no se soportan, puede ser muy difícil y costoso migrar la aplicación a una nueva tecnología.

**Evolución e integración de nuevas funcionalidades**

- Rigidez en la incorporación de nuevas funcionalidades: Una de las principales limitaciones de las aplicaciones monolíticas es su rigidez para incorporar nuevas funcionalidades. Debido a su naturaleza monolítica, cualquier cambio o adición puede tener efectos en cascada que afectan a múltiples componentes del sistema. Por ejemplo, si se quiere añadir una nueva funcionalidad que requiere una nueva dependencia, ésta puede entrar en conflicto con las dependencias existentes y causar problemas en otras partes de la aplicación. Este riesgo puede hacer que los desarrolladores sean reacios a añadir nuevas funcionalidades, lo que limita la capacidad de la aplicación para evolucionar y adaptarse a las cambiantes necesidades del negocio.

- Riesgo de interrupción del servicio: El proceso de despliegue de una aplicación monolítica puede ser lento y costoso. Dado que toda la aplicación se ejecuta como una única unidad, cualquier actualización requiere que se despliegue la aplicación completa. Esto puede llevar a tiempos de inactividad prolongados, durante los cuales la aplicación

no está disponible para los usuarios. Además, si algo sale mal durante el despliegue, puede causar una interrupción total del servicio. Este riesgo de interrupción puede ser particularmente problemático para aplicaciones críticas que requieren una alta disponibilidad.

**Flexibilidad y adaptabilidad**

Las aplicaciones monolíticas a menudo se construyen utilizando un solo lenguaje de programación o una pila tecnológica específica. Esto puede resultar en una falta de flexibilidad para adoptar nuevas tecnologías o incorporar nuevos servicios. Por ejemplo, si surge una nueva tecnología que puede mejorar una parte específica de la aplicación, puede ser difícil o incluso imposible adoptarla sin reescribir toda la aplicación.

Además, la integración de servicios externos puede ser más complicada en una arquitectura monolítica. Los cambios en los puntos de integración pueden requerir modificaciones significativas en el código de la aplicación, y los problemas de integración pueden tener efectos de gran alcance que afecten a toda la aplicación.

En contraste, en una arquitectura basada en microservicios o serverless, cada servicio puede utilizar la tecnología que mejor se adapte a sus necesidades, y los servicios pueden ser modificados, reemplazados o escalados de manera independiente. Además, las fallas en un servicio no tienen por qué afectar a los demás, lo que mejora la resiliencia de la aplicación en su conjunto.

Este nivel de flexibilidad puede ser particularmente valioso en un entorno empresarial en constante cambio, donde la capacidad de adaptarse rápidamente a las nuevas tecnologías y tendencias puede ser una ventaja competitiva importante. Sin embargo, también viene con su propio conjunto de desafíos, como la necesidad de coordinar efectivamente entre múltiples servicios y mantener la coherencia en toda la aplicación.

Las limitaciones de las aplicaciones monolíticas en términos de escalabilidad, mantenimiento, evolución e integración de nuevas funcionalidades pueden hacer que las aplicaciones monolíticas sean menos eficientes y más costosas de operar a gran escala. Además, pueden limitar la capacidad de una empresa para responder rápidamente a los cambios en la

demanda, lo que puede ser un desafío en el entorno empresarial actual, donde las cargas de trabajo pueden cambiar rápidamente y de manera impredecible.

Para superar estos desafíos, es necesario explorar enfoques basados en arquitecturas de microservicios y serverless, que permitan una mayor flexibilidad y adaptabilidad en la gestión de aplicaciones empresariales.

### c. Estrategia de modularización y ecosistema de aplicaciones

En una estrategia de modularización y construcción de un ecosistema de aplicaciones, el objetivo es descomponer la aplicación monolítica en componentes más pequeños e independientes, cada uno con su propio conjunto de responsabilidades y reglas de negocio. Este enfoque puede mejorar la escalabilidad, la flexibilidad y la capacidad de mantenimiento de la aplicación, así como facilitar la incorporación de nuevas funcionalidades y tecnologías.

Para implementar esta estrategia, es útil seguir una serie de pasos que ayuden a guiar el proceso:

1. Identificar los módulos: El primer paso es analizar la aplicación existente y dividir sus funcionalidades en módulos lógicos. Estos módulos deben ser lo más independientes posible, con un dominio claramente definido y una interfaz bien diseñada para interactuar con otros módulos. Por ejemplo, en una aplicación de comercio electrónico, se pueden identificar módulos como inventario, pedidos, pagos y autenticación de usuarios.
2. Diseñar la comunicación: Una vez identificados los módulos, se debe definir cómo interactuarán entre sí y con el sistema en general. Esto puede incluir la elección de protocolos de comunicación, como HTTP o gRPC, y la implementación de APIs para facilitar la interacción entre los módulos.
3. Desacoplar la base de datos: En muchas aplicaciones monolíticas, todas las funcionalidades comparten una única base de datos. En una arquitectura de microservicios, cada módulo debería tener su propia base de datos, o al menos su propio esquema dentro de una base de datos compartida. Esto permite una mayor flexibilidad y escalabilidad a nivel de base de datos, y reduce las posibles dependencias y conflictos entre los módulos.
4. Implementar microservicios: Una vez definidos los módulos y diseñada la comunicación, se puede comenzar a convertir cada

módulo en un microservicio independiente. Esto puede implicar la adopción de tecnologías de contenedores, como Docker o Kubernetes, o plataformas serverless, como AWS Lambda o Azure Functions.

5. Integrar funcionalidades serverless: Algunas partes de la aplicación pueden beneficiarse especialmente de la adopción de la computación serverless, como las funciones que responden a eventos específicos o que realizan tareas de procesamiento intensivo. Estas funciones pueden implementarse como servicios serverless, lo que puede aumentar la eficiencia y reducir la carga operativa.
6. Monitoreo y gestión: A medida que la aplicación se divide en múltiples microservicios y funciones serverless, se vuelve cada vez más importante contar con mecanismos de monitoreo y gestión centralizados. Esto puede incluir herramientas de monitoreo de rendimiento, como New Relic o Datadog, y sistemas de gestión de logs y trazas, como ELK Stack o Splunk.
7. Despliegue y entrega continua: Finalmente, se debe implementar un proceso de integración y entrega continua (CI/CD) que permita desplegar y actualizar cada microservicio y función serverless de forma rápida y segura. Esto puede implicar el uso de herramientas como Jenkins, CircleCI, o GitLab CI/CD, y puede incluir prácticas como las pruebas

En este diagrama, la aplicación monolítica se representa con varios módulos interactuando con una base de datos compartida.

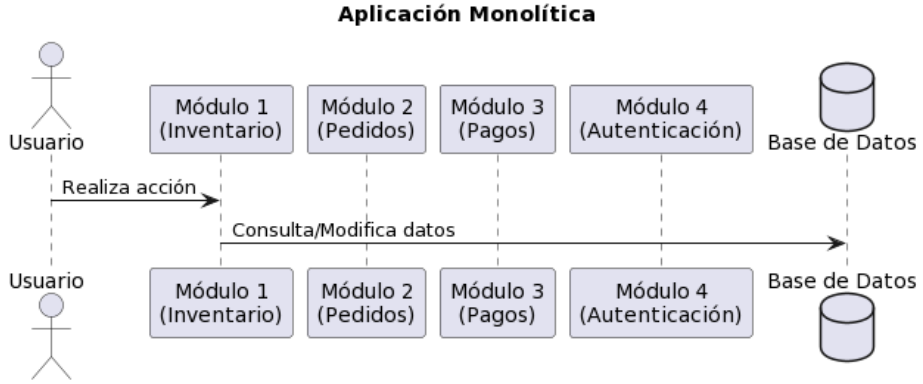



Ilustración 1 Aplicación Monolítica

En este diagrama, la aplicación después de la migración se representa con cada microservicio interactuando con su propia base de datos y las funciones serverless integradas a través de un API Gateway.

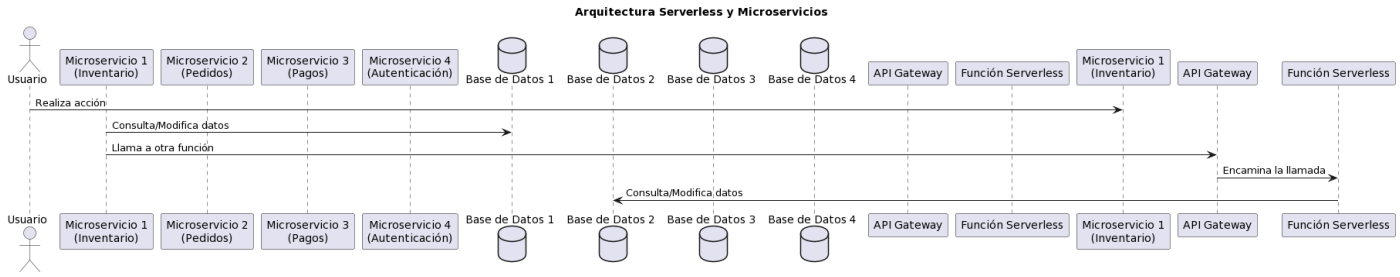


Ilustración 2 Aplicación modularizada en microservicios

## 3. Arquitecturas serverless y gestión de APIs en la transformación empresarial

### a. Introducción

Las APIs (Application Programming Interfaces) son un componente esencial de los sistemas modernos. Permiten que diferentes aplicaciones y sistemas se comuniquen entre sí, lo que es fundamental para el intercambio de información y la integración de sistemas.

La infraestructura y gestión de las APIs es un desafío importante para las empresas. A medida que las empresas adoptan más APIs, se enfrentan a la necesidad de gestionar un número creciente de ellas. Esto puede ser difícil de mantener y escalar, y también puede plantear problemas de seguridad y control de acceso.

En este documento, exploraremos el papel de las arquitecturas serverless en la gestión de APIs. Las arquitecturas serverless ofrecen una serie de ventajas para la gestión de APIs, incluyendo la escalabilidad, la seguridad y la facilidad de uso.

La gestión de APIs ha evolucionado a lo largo del tiempo para adaptarse a las necesidades cambiantes de las empresas. En el pasado, las empresas solían implementar sus propias APIs en servidores locales. Esto requería una inversión significativa en infraestructura y personal.

Con el auge de la computación en la nube, las empresas comenzaron a adoptar un enfoque más descentralizado para la gestión de APIs. Esto supuso la adopción de servicios de API en la nube, como los gateways de API y las plataformas de gestión de API.

Los servicios de API en la nube ofrecen una serie de ventajas sobre la implementación de APIs en servidores locales. Son más escalables, fáciles de mantener y más seguros.

Actualmente existen una serie de iniciativas y estándares que buscan mejorar la gestión de las API. El Consorcio de Interconexión de Servicios (Service Interoperability Consortium, o SIC) desarrolló el estándar Service Component Architecture (SCA) para simplificar el desarrollo de aplicaciones de servicios. El Consorcio Abierto de Aplicaciones Web (Open Web Application Security Project, o OWASP) proporciona orientación y recursos para garantizar la seguridad de las API.

Las arquitecturas serverless son una tendencia creciente en la gestión de API y en los sistemas modernos. La idea detrás del enfoque serverless es liberar a los programadores de las preocupaciones relacionadas con la administración de servidores. De esta manera, los proveedores de servicios en la nube se encargan de la infraestructura, lo que permite a los desarrolladores centrarse en la creación de aplicaciones y funcionalidades sin tener que pensar en la escalabilidad y el mantenimiento de los servidores.

Esta forma de trabajar también facilita la gestión de las API, porque los desarrolladores pueden crear y desplegar funciones individuales que se activan según eventos o solicitudes de API específicas. Este enfoque modular hace que la administración de las API sea más sencilla en comparación con los sistemas monolíticos.

Además, las arquitecturas serverless pueden contribuir a solucionar problemas de seguridad y control de acceso en la gestión de API. Como las funciones serverless se ejecutan en entornos separados, hay menos riesgo de

que un atacante pueda acceder a recursos compartidos o comprometer todo el sistema. También es posible establecer políticas de control de acceso y autorización para cada función, lo que proporciona un mayor control en la administración de permisos.

Este tipo de arquitecturas serverless ofrecen una serie de ventajas para la gestión de APIs, incluyendo:

- Escalabilidad: Las APIs serverless se escalan automáticamente según la demanda, lo que elimina la necesidad de gestionar la capacidad de los servidores.
- Seguridad: Las funciones serverless se ejecutan en entornos aislados, lo que reduce el riesgo de ataques.
- Facilidad de uso: Las APIs serverless son fáciles de crear y desplegar, lo que reduce el tiempo y los costes de desarrollo.

A pesar de las ventajas de las arquitecturas serverless, es importante tener en cuenta algunas consideraciones para la gestión de APIs en este contexto:

- Gestión de eventos: Las APIs serverless se activan por eventos, lo que puede dificultar el seguimiento y la depuración de los problemas.
- Gestión de costos: Los costos de las APIs serverless se basan en el uso, por lo que es importante supervisar el uso para evitar costos excesivos.
- Gestión de la seguridad: Las APIs serverless son seguras por defecto, pero es importante aplicar las medidas de seguridad adecuadas para proteger los datos y los sistemas.

### b. Aplicaciones Serverless

El cómputo sin servidor (serverless) ha experimentado un rápido crecimiento en la industria de la computación en la nube gracias a sus ventajas en cuanto a simplicidad, escalabilidad y costo. A continuación, se presenta un estado del arte sobre el cómputo sin servidor y se profundiza en los conceptos relacionados.

Función como Servicio (FaaS): FaaS es un modelo de implementación en el que los desarrolladores crean y ejecutan aplicaciones en forma de funciones

individuales y autónomas. Estas funciones se activan mediante eventos y se ejecutan en respuesta a ellos. El cómputo sin servidor se basa en este modelo, lo que permite a los desarrolladores centrarse en la lógica de las aplicaciones sin preocuparse por la infraestructura subyacente.

Backend como Servicio (BaaS): BaaS es un enfoque en el que los servicios de backend, como bases de datos y almacenamiento de objetos, se ofrecen como servicios gestionados en la nube. Estos servicios se integran con las aplicaciones basadas en FaaS, lo que permite a los desarrolladores centrarse en el código de la aplicación mientras se benefician de las ventajas de escalabilidad y rendimiento de los servicios gestionados.

**Retos y oportunidades en el cómputo sin servidor**

A pesar de sus ventajas, el cómputo sin servidor enfrenta una serie de desafíos, como la latencia de inicio, la gestión de recursos y la adaptación a sus requerimientos específicos. Para superar estos desafíos, se han propuesto numerosas investigaciones y enfoques en áreas como:

- Optimización de la latencia de inicio: La latencia de inicio es el tiempo que tarda una función en comenzar a ejecutarse después de ser invocada. La latencia puede ser un factor crítico en aplicaciones sensibles al tiempo. Se han propuesto técnicas como el precalentamiento de instancias y la reutilización de contenedores para reducir la latencia de inicio en las plataformas sin servidor.

- Gestión eficiente de recursos: El cómputo sin servidor se basa en la asignación dinámica de recursos para ejecutar funciones. Los algoritmos de programación y asignación de recursos eficientes son esenciales para garantizar un rendimiento óptimo y reducir el desperdicio de recursos. Se han propuesto enfoques basados en aprendizaje automático y optimización para mejorar la gestión de recursos en entornos sin servidor.

- Seguridad y privacidad: La seguridad y la privacidad son preocupaciones clave en el cómputo sin servidor, ya que las funciones pueden ejecutarse en un entorno compartido y ser propensas a ataques. Se han propuesto técnicas de aislamiento,

cifrado y monitoreo para garantizar la seguridad y la privacidad de los datos y las aplicaciones en entornos sin servidor.

- Integración con otras tecnologías: El cómputo sin servidor se ha integrado con éxito en diversas áreas, como el edge computing, el Internet de las cosas (IoT) y la inteligencia artificial (IA). Estas integraciones permiten aprovechar las ventajas del cómputo sin servidor en contextos específicos y ofrecen oportunidades para nuevas investigaciones y aplicaciones.

Si miramos la evolución de la virtualización de la computación, ésta ha pasado por tres etapas: Máquinas Virtuales (VM), Contenedores y Serverless.

- Máquinas Virtuales (VM): Las máquinas virtuales (VM) son una capa de abstracción que permiten ejecutar múltiples sistemas operativos (OS) en una única máquina física. Esto se logra mediante el uso de un software llamado hipervisor, que emula el hardware físico y crea un entorno virtualizado en el que se pueden ejecutar varios OS de forma aislada. Las VMs proporcionan un alto grado de aislamiento y una mejor utilización de los recursos de hardware. Sin embargo, también tienen una sobrecarga de memoria considerable debido a la necesidad de ejecutar sistemas operativos completos, lo que puede afectar el rendimiento del sistema en general.

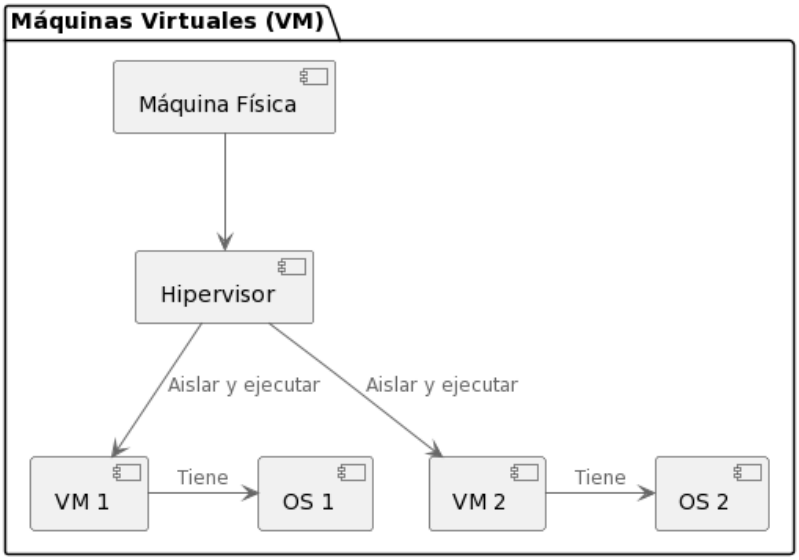



Ilustración 3. Esquema arquitectura VM

- Contenedores: Para abordar las limitaciones de las máquinas virtuales, los contenedores surgieron como una solución más ligera y eficiente. Los contenedores permiten la virtualización a nivel del sistema operativo, lo que significa que comparten el mismo kernel del sistema operativo, pero mantienen un aislamiento entre aplicaciones y sus dependencias. Docker es una de las tecnologías de contenedores más populares, que facilita la creación, implementación y ejecución de aplicaciones en contenedores. Los contenedores son más rápidos de implementar y ofrecen mayor portabilidad, lo que reduce los costos y el uso de recursos en comparación con las VMs.

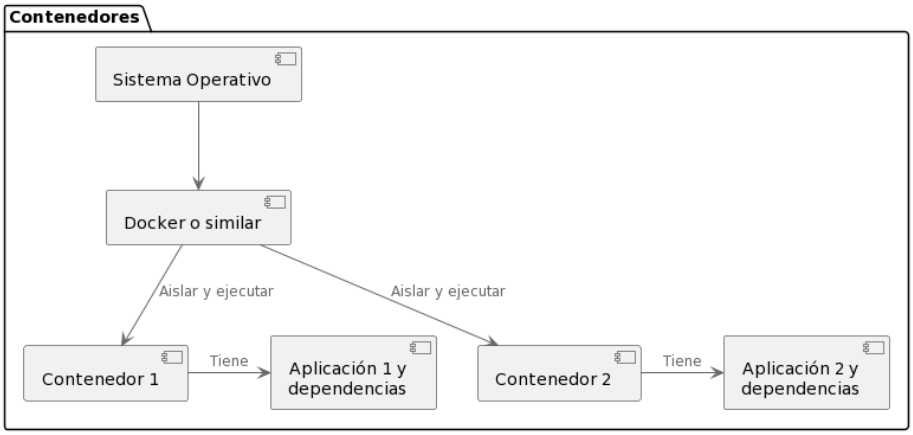



Ilustración 4. Esquema arquitectura de Contenedores

- Serverless: La computación serverless es la siguiente etapa en la evolución de la virtualización. A diferencia de las VMs y los contenedores, las aplicaciones serverless no requieren que los desarrolladores administren directamente la infraestructura subyacente. En su lugar, las plataformas serverless, como AWS Lambda, manejan automáticamente el aprovisionamiento y la escala de los recursos necesarios para ejecutar aplicaciones en función de la demanda. Los desarrolladores solo necesitan escribir y desplegar funciones, que luego son ejecutadas por la plataforma serverless según sea necesario. Esto simplifica enormemente el desarrollo y la administración, ya que los desarrolladores no tienen que preocuparse por la gestión de recursos y configuraciones complejas. Además, las plataformas serverless suelen seguir un modelo de facturación basado en el uso, lo que permite una utilización más eficiente de los recursos y menores costos.

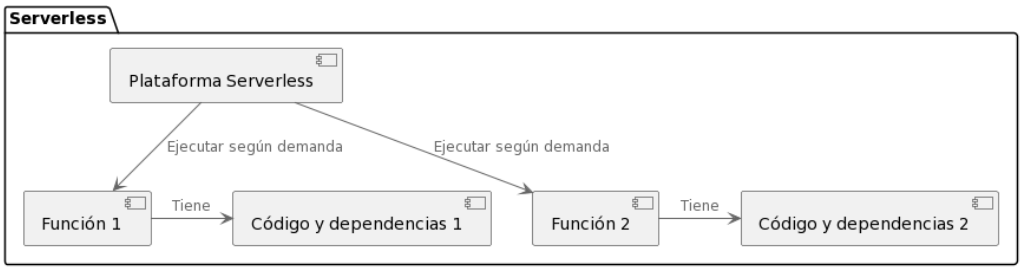



Ilustración 5. Esquema arquitectura Serverless

Se constata así que las aplicaciones serverless son más ligeras y flexibles, y se centran en reducir la carga de gestión de recursos y configuraciones complejas para los desarrolladores. La plataforma serverless actúa como mediador entre la aplicación y la infraestructura subyacente, simplificando el desarrollo y la administración.

Existen una serie características clave de la computación serverless, que suponen tanto retos como oportunidades a la hora de diseñar una aplicación serverless:

- Sin host y elástica: los usuarios no necesitan trabajar con servidores y se benefician de una menor carga operativa.
- Ligereza: las aplicaciones serverless suelen tener pocas funciones y permiten a los desarrolladores centrarse en la lógica única de la aplicación.
- Sin estado: las funciones serverless se ejecutan en contenedores sin estado, lo que facilita la escalabilidad, pero limita las aplicaciones que dependen de la transferencia de información de estado.
- Tiempos de ejecución cortos pero variables: las funciones serverless suelen estar activas durante periodos cortos de tiempo, pero la granularidad de la facturación sigue siendo un desafío.
- Paralelismo intrafunción deficiente: el paralelismo dentro de una sola función es limitado, lo que puede afectar la eficiencia en ciertos casos.
- Migración: las funciones serverless tienen un coste de inicio, lo que puede aumentar la latencia y la sobrecarga del sistema durante el auto escalado y el equilibrio de carga.
- Contenedores separados: la comunicación entre contenedores separados en aplicaciones serverless puede generar mayor latencia y presión en la red.
- Explosividad: las cargas de trabajo en la computación serverless suelen ser explosivas, lo que plantea desafíos para la planificación de tareas.

A pesar de los beneficios y la popularidad de los sistemas serverless, diseñar una plataforma serverless que cumpla con los requisitos de eficiencia y escalabilidad sigue siendo un desafío importante.

### c. Casos de uso y diferentes alternativas en el mercado.

**Amazon Web Services (AWS)**

Amazon Web Services (AWS) es un conjunto integral de servicios en la nube que proporciona una plataforma altamente escalable, fiable y segura para el desarrollo y despliegue de aplicaciones. Desde su lanzamiento en 2006, AWS ha experimentado un rápido crecimiento y se ha convertido en líder del mercado en soluciones de computación en la nube.

AWS ofrece una amplia gama de servicios que abarcan múltiples áreas, como la computación, el almacenamiento, las bases de datos, el análisis, la inteligencia artificial, la seguridad y el Internet de las Cosas (IoT). Estos servicios se agrupan en categorías que facilitan su selección y utilización por parte de los desarrolladores y las empresas.

La infraestructura global de AWS está dividida en regiones, que a su vez constan de múltiples zonas de disponibilidad (AZ). Las regiones son áreas geográficas que contienen al menos dos zonas de disponibilidad, mientras que las zonas de disponibilidad son centros de datos independientes que garantizan la continuidad del servicio en caso de fallos o interrupciones. Esta arquitectura permite a los usuarios diseñar aplicaciones resilientes y tolerantes a fallos, así como optimizar la latencia y la transferencia de datos.

Uno de los principales beneficios de AWS es su modelo de pago por uso, que permite a los usuarios pagar solo por los recursos que consumen, sin compromisos a largo plazo ni inversiones iniciales. Este enfoque reduce los costes y proporciona una mayor flexibilidad para adaptarse a las cambiantes necesidades empresariales.

Algunos casos de uso y servicios serverless populares en AWS son:

- AWS Lambda: Es el servicio de computación serverless más utilizado en AWS. Permite ejecutar código en respuesta a eventos, como cambios en datos, solicitudes de usuario o eventos de otros servicios de AWS. Los casos de uso típicos incluyen procesamiento de datos en tiempo real, transformación de archivos, tareas de fondo y sistemas de cola.

- API Gateway: Proporciona una forma de crear, publicar y mantener APIs RESTful y WebSocket para aplicaciones serverless. Puedes usar API Gateway junto con Lambda para crear APIs escalables y sin servidor.

- Amazon S3: Es un servicio de almacenamiento de objetos que puede almacenar y recuperar grandes volúmenes de datos. Se puede utilizar en combinación con Lambda para procesar y transformar archivos almacenados en S3.

- Amazon DynamoDB: Es una base de datos NoSQL completamente administrada que ofrece un rendimiento rápido y predecible con escalabilidad automática. Puedes utilizar DynamoDB en aplicaciones serverless para almacenar y recuperar datos sin preocuparte por la administración de la infraestructura.

- AWS Step Functions: Permite coordinar y orquestar flujos de trabajo serverless en aplicaciones de múltiples etapas. Puedes utilizar Step Functions para diseñar y ejecutar flujos de trabajo que coordinen servicios como Lambda, DynamoDB y S3.

- Amazon EventBridge: Es un servicio de bus de eventos serverless que facilita la conexión de aplicaciones utilizando eventos basados en datos. Puedes utilizar EventBridge para enrutar eventos desde fuentes como SaaS, aplicaciones personalizadas y servicios de AWS a objetivos como funciones Lambda y otros servicios de AWS.

- AWS App Runner: Es un servicio completamente administrado que facilita la creación, implementación y escalado de aplicaciones de contenedores rápidamente. Puedes utilizar App Runner para implementar aplicaciones serverless desde imágenes de contenedor o código fuente directamente.

**Microsoft Azure**

Microsoft Azure es otra plataforma en la nube líder que ofrece soluciones serverless. Al igual que AWS, Azure proporciona servicios que permiten a los

desarrolladores construir y ejecutar aplicaciones sin preocuparse por la gestión de la infraestructura subyacente.

Azure proporciona una variedad de servicios en áreas como computación, almacenamiento, bases de datos, análisis, inteligencia artificial, seguridad e Internet de las Cosas (IoT).

La infraestructura global de Azure está compuesta por múltiples regiones geográficas, cada una de las cuales contiene al menos dos zonas de disponibilidad (AZ) para garantizar la resiliencia y continuidad del servicio. Esto permite a los usuarios diseñar aplicaciones tolerantes a fallos y optimizar la latencia y el rendimiento.

A continuación, se presentan algunos de los principales servicios serverless y casos de uso en Azure:

- Azure Functions: Es el equivalente de AWS Lambda en la plataforma Azure. Permite a los desarrolladores escribir y ejecutar código en respuesta a eventos, como solicitudes HTTP, cambios en la base de datos o mensajes en colas. Los casos de uso comunes incluyen procesamiento de datos en tiempo real, tareas de fondo, sistemas de cola y microservicios.

- Azure Logic Apps: Es un servicio que permite crear y ejecutar flujos de trabajo basados en lógica que se integran con diferentes servicios, tanto en la nube como local. Los desarrolladores pueden utilizar Logic Apps para orquestar y automatizar procesos empresariales y de integración de aplicaciones.

- Azure API Management: Es un servicio que ayuda a los desarrolladores a crear, publicar, mantener y proteger APIs. Puede utilizarse en combinación con Azure Functions para crear y administrar APIs serverless.

- Azure Cosmos DB: Es una base de datos NoSQL completamente administrada y distribuida globalmente que ofrece escalabilidad y rendimiento en tiempo real. Puede utilizarse en aplicaciones serverless para almacenar y recuperar datos con latencia extremadamente baja y alta disponibilidad.

- Azure Event Grid: Es un servicio de enrutamiento de eventos que permite conectar fuentes de eventos y suscriptores mediante publicación-suscripción. Puede utilizarse en combinación con Azure Functions para desencadenar la ejecución de código en respuesta a eventos específicos.

- Azure Service Bus: Es un servicio de mensajería basado en la nube que permite la comunicación entre aplicaciones y servicios a través de mensajes. Puede utilizarse para conectar aplicaciones serverless y permitir la comunicación asíncrona entre ellas.

- Azure Kubernetes Service (AKS): Es un servicio de orquestación de contenedores que facilita la implementación, administración y escalado de aplicaciones basadas en contenedores. Aunque no es inherentemente serverless, puede combinarse con la funcionalidad Virtual Kubelet y el proyecto KEDA para habilitar características serverless en aplicaciones basadas en contenedores.

**Google Cloud Platform (GCP)**

Google Cloud Platform (GCP) es otra plataforma en la nube líder que ofrece soluciones serverless para facilitar el desarrollo y la implementación de aplicaciones sin preocuparse por la gestión de la infraestructura subyacente. Desde su lanzamiento en 2011, GCP ha experimentado un crecimiento significativo y se ha convertido en un actor relevante en el mercado de la nube

GCP proporciona servicios en diversas áreas, como computación, almacenamiento, bases de datos, análisis de datos, inteligencia artificial y aprendizaje automático. Estos servicios están diseñados para ser fácilmente accesibles y utilizables por empresas y desarrolladores

Algunos de los principales servicios serverless y casos de uso en GCP incluyen:

- Cloud Functions: Es el equivalente de AWS Lambda y Azure Functions en GCP. Permite a los desarrolladores escribir y ejecutar código en respuesta a eventos específicos, como solicitudes HTTP, cambios en la base de datos o mensajes en colas. Los casos de uso

comunes incluyen procesamiento de datos en tiempo real, tareas de fondo y microservicios.

- Cloud Run: Es un servicio que permite a los desarrolladores implementar y escalar contenedores sin preocuparse por la gestión de la infraestructura. A diferencia de Cloud Functions, que se centra en ejecutar funciones individuales, Cloud Run permite implementar aplicaciones completas en contenedores.

- App Engine: Es un servicio de plataforma como servicio (PaaS) que permite a los desarrolladores crear, implementar y escalar aplicaciones en la nube sin preocuparse por la gestión de la infraestructura. App Engine admite múltiples lenguajes de programación, incluidos Python, Java, Node.js, Go y PHP.

- Firestore: Es una base de datos NoSQL completamente administrada que permite a los desarrolladores almacenar y recuperar datos para aplicaciones serverless. Firestore proporciona escalabilidad, rendimiento y latencia en tiempo real, similar a Azure Cosmos DB y AWS DynamoDB.

- Pub/Sub: Es un servicio de mensajería basado en la nube que permite la comunicación entre aplicaciones y servicios a través de mensajes. Puede utilizarse para conectar aplicaciones serverless y permitir la comunicación asíncrona entre ellas.

- API Gateway: Es un servicio que ayuda a los desarrolladores a crear, publicar, mantener y proteger APIs. Puede utilizarse en combinación con Cloud Functions o Cloud Run para crear y administrar APIs serverless.

- Cloud Scheduler: Es un servicio de programación de trabajos basado en la nube que permite a los desarrolladores programar tareas recurrentes o temporizadas. Puede utilizarse para desencadenar funciones serverless en momentos específicos o según un cronograma.

- Cloud Workflows: Es un servicio que permite a los desarrolladores crear y ejecutar flujos de trabajo basados en lógica que se integran

con diferentes servicios en la nube. Los desarrolladores pueden utilizar Cloud Workflows para orquestar y automatizar procesos empresariales y de integración de aplicaciones.

**Tablas Comparativas**

Tabla comparativa general de servicios serverless

| Característica | AWS | Azure | GCP |
|---|---|---|---|
| **Funciones Serverless** | Lambda | Functions | Cloud Functions |
| **Almacenamiento Serverless** | S3 | Blob Storage | Cloud Storage |
| **Bases de datos Serverless** | DynamoDB, Aurora Serverless | Cosmos DB | Firestore |
| **Mensajería y Eventos Serverless** | EventBridge, SNS, SQS | Event Grid, Service Bus, Event Hubs | Cloud Pub/Sub |
| **API Gateway** | API Gateway | API Management | Cloud Endpoints, Apigee |
| **Orquestación de flujo de trabajo** | Step Functions | Logic Apps | Cloud Workflows |
| **Transmisión de datos en tiempo real** | Kinesis Data Streams | Event Hubs | Datastream |

Tabla comparativa de Funciones de cómputo serverless

| Característica | AWS Lambda | Azure Functions | Google Cloud Functions |
|---|---|---|---|
| **Lenguajes soportados** | Python, Node.js, Java, C#, Go, Ruby, PowerShell | C#, Java, JavaScript, Python, PowerShell, TypeScript | Node.js, Python, Go, Java, .NET, Ruby, PHP |
| **Modelo de facturación** | Por invocación, duración y recursos utilizados | Por invocación, duración y recursos utilizados | Por invocación, duración y recursos utilizados |
| **API Gateway** | Amazon API Gateway | Azure API Management | Google Cloud Endpoints |
| **Almacenamiento** | Amazon S3, Amazon DynamoDB, Amazon | Azure Blob Storage, Azure Cosmos DB, | Google Cloud Storage, Firestore, Cloud SQL |

| | | | |
|---|---|---|---|
| | RDS | Azure SQL | |
| **Integraciones** | AWS Step Functions, SNS, SQS, EventBridge, etc. | Azure Logic Apps, Event Grid, Service Bus, etc. | Cloud Pub/Sub, Cloud Scheduler, Cloud Tasks, etc. |
| **Escalabilidad** | Automática | Automática | Automática |
| **Tiempo de vida máximo** | 15 minutos | Sin límite definido | 9 minutos |
| **Monitoreo** | Amazon CloudWatch | Azure Monitor | Stackdriver Logging y Monitoring |

### d. Comparativa de serverless versus infraestructuras con servidor

Al comparar las arquitecturas serverless con las infraestructuras con servidor, es crucial centrarse en sus ventajas y desventajas, así como en los casos de uso adecuados de cada paradigma.

A continuación, se describen los pros y contras de cada enfoque y se presentan los casos de uso ideales para ambas arquitecturas.

**Serverless:**

Pros:

- Menor costo total de propiedad: Debido a que sólo se paga por el tiempo de ejecución real de la función, en lugar de tener que pagar por recursos de servidor inactivos, el costo total de propiedad puede ser significativamente menor con la arquitectura serverless, especialmente para aplicaciones con patrones de tráfico impredecibles o intermitentes.

- Escalabilidad automática: Las funciones serverless pueden escalar automáticamente para manejar el tráfico entrante, sin necesidad de intervención manual. Esto puede ser de gran beneficio para las aplicaciones que experimentan picos de tráfico.

- Menor tiempo de desarrollo y enfoque en la lógica de negocio: Con serverless, los desarrolladores pueden concentrarse en escribir código para la lógica de negocio, en lugar de tener que preocuparse por la gestión de la infraestructura, lo que puede acelerar el tiempo de desarrollo.

Contras:

- Menor control sobre la infraestructura: Debido a que la administración de la infraestructura es manejada por el proveedor de la nube, los desarrolladores tienen menos control sobre la infraestructura en la que se ejecutan sus aplicaciones.

- Latencia potencialmente mayor: Para algunas aplicaciones, especialmente las que requieren tiempos de respuesta muy rápidos, la latencia adicional introducida por el inicio en frío de las funciones serverless puede ser un problema.

- Limitaciones en la personalización y las configuraciones: Las funciones serverless pueden tener limitaciones en términos de personalización y configuración, como el tiempo de ejecución máximo de una función, el tamaño máximo de despliegue, etc.

Dependiendo de los requisitos específicos de la aplicación y las restricciones, la arquitectura serverless puede ser una opción más atractiva que la infraestructura con servidor. Sin embargo, no es una solución de talla única, y hay casos en los que una infraestructura con servidor puede ser más adecuada. Por ejemplo, las aplicaciones que requieren un control más detallado sobre la infraestructura, o las que tienen requisitos de latencia muy bajos, pueden beneficiarse más de una infraestructura con servidor.

Casos de uso adecuados para serverless:

- Aplicaciones web y móviles: Las arquitecturas serverless son ideales para aplicaciones web y móviles que requieren escalabilidad y rápida implementación, como sitios web de comercio electrónico o aplicaciones de redes sociales.

- Sistemas de análisis de datos en tiempo real: Los servicios serverless pueden procesar y analizar grandes volúmenes de datos en tiempo real, lo que es útil en casos como el monitoreo de tráfico web o la detección de fraudes en transacciones financieras.
- Aplicaciones de Internet de las cosas (IoT): Las soluciones serverless pueden manejar eficientemente la ingesta y procesamiento de datos de dispositivos IoT, lo que permite la creación de aplicaciones y servicios innovadores en este ámbito.

**Infraestructuras con servidor:**

Pros:

- Mayor control y personalización: Las infraestructuras con servidor proporcionan un mayor control sobre el entorno de ejecución. Los desarrolladores pueden elegir el sistema operativo, las configuraciones de hardware y software, y controlar aspectos detallados del entorno de ejecución.

- Adecuado para aplicaciones empresariales críticas: Las aplicaciones empresariales críticas que requieren altos niveles de seguridad, cumplimiento y rendimiento constante pueden beneficiarse del control detallado que ofrece una infraestructura con servidor.

- Adecuado para sistemas de procesamiento intensivo: Las tareas que requieren una gran cantidad de recursos de CPU, memoria o almacenamiento pueden ejecutarse de manera más eficiente en una infraestructura con servidor, donde los recursos pueden ser personalizados y optimizados para la tarea en cuestión.

Contras:

- Mayor costo total de propiedad: Las infraestructuras con servidor pueden tener un costo total de propiedad más alto debido a los costos de administración y mantenimiento de la infraestructura.

- Gestión de recursos y configuración manual: Los desarrolladores deben gestionar y configurar manualmente los recursos, lo que puede aumentar la complejidad y el tiempo de desarrollo.

- Escalabilidad manual: A diferencia de las arquitecturas serverless, las infraestructuras con servidor requieren que los desarrolladores o administradores escalen manualmente los recursos para adaptarse a los cambios en la demanda, lo que puede ser un proceso desafiante y que consume tiempo.

Al igual que con la arquitectura serverless, la elección de usar una infraestructura con servidor dependerá de los requisitos específicos de la aplicación y de los recursos disponibles. En algunos casos, una infraestructura con servidor puede ser la opción más adecuada, mientras que en otros, una arquitectura serverless puede ser más eficiente y rentable.

Casos de uso adecuados para infraestructuras con servidor:

- Aplicaciones empresariales críticas: Las infraestructuras con servidor brindan el control y la personalización necesarios para aplicaciones empresariales críticas, como sistemas de gestión de recursos empresariales (ERP) o sistemas de planificación de la cadena de suministro.
- Sistemas de procesamiento intensivo: Las infraestructuras con servidor son ideales para aplicaciones y sistemas que requieren una gran cantidad de recursos informáticos y de almacenamiento, como renderizado de gráficos en 3D o simulaciones científicas.
- Aplicaciones que requieren personalización y control granular: Las aplicaciones que necesitan un alto grado de personalización en la configuración y gestión de la infraestructura, como plataformas de videojuegos en línea o aplicaciones de realidad virtual, pueden beneficiarse de la flexibilidad que ofrecen las infraestructuras con servidor.

Al comparar las arquitecturas serverless con las infraestructuras con servidor, es esencial centrarse en las ventajas y desventajas de cada enfoque y en los casos de uso adecuados. Mientras que las arquitecturas serverless son ideales para aplicaciones que requieren escalabilidad y rápida implementación, las infraestructuras con servidor son más adecuadas para aplicaciones empresariales críticas y sistemas de procesamiento intensivo.

### e. API Management sobre serverless: Concepto y beneficios.

La gestión de API (Application Programming Interface) se ha convertido en un componente esencial en el diseño y desarrollo de arquitecturas modernas de software. Este enfoque permite a las organizaciones exponer sus servicios y datos de manera segura y estandarizada, facilitando la integración entre aplicaciones y sistemas. En particular, las arquitecturas serverless se han vuelto cada vez más populares debido a su capacidad para escalar automáticamente y reducir el tiempo de desarrollo. Este punto profundiza en el estado del arte, los beneficios y las perspectivas de la gestión de API en arquitecturas serverless.

La gestión de API en arquitecturas serverless se ha expandido rápidamente en los últimos años, impulsada por la creciente demanda de soluciones escalables y elásticas que faciliten la interacción entre servicios y aplicaciones. Algunas de las tecnologías y enfoques clave en este ámbito incluyen:

**Plataformas de API Gateway:** Estas soluciones proporcionan un punto de entrada único para todas las API, facilitando la autenticación, autorización, enrutamiento, transformación y monitoreo de las solicitudes. Ejemplos de plataformas populares incluyen Amazon API Gateway, Azure API Management y Google Cloud API Gateway.

Estas alternativas serán analizadas y comparadas en el punto siguiente.

**Especificaciones y Estándares:** La adopción de especificaciones y estándares como OpenAPI y JSON Web Tokens (JWT) ha permitido la creación de API más interoperables y seguras, lo que facilita su integración en arquitecturas serverless.

**OpenAPI**, anteriormente conocido como Swagger, es una especificación para la definición de APIs. Permite documentar y definir todos los aspectos de una API REST, incluyendo puntos de acceso, parámetros de solicitud, respuestas, encabezados y autenticación.

Es una especificación para describir APIs RESTful. Es un formato estándar legible por máquina que proporciona una descripción completa de una API, incluyendo su estructura, sus operaciones y sus parámetros.

OpenAPI se basa en el lenguaje de marcado YAML, que es un formato de texto ligero y fácil de leer. Esto hace que OpenAPI sea fácil de comprender y de trabajar con él.

OpenAPI ofrece una serie de ventajas para la creación y el uso de APIs:

- Interoperabilidad: OpenAPI permite que las APIs se describan de una forma estandarizada, lo que facilita su uso por parte de otros desarrolladores.
- Documentación: OpenAPI puede utilizarse para generar documentación para las APIs, lo que facilita su comprensión por parte de los usuarios.
- Pruebas: OpenAPI puede utilizarse para generar casos de prueba para las APIs, lo que facilita su comprobación y su desarrollo.
- Autogeneración: OpenAPI puede utilizarse para generar código para las APIs, lo que facilita su desarrollo.

OpenAPI es una especificación ampliamente adoptada por la comunidad de desarrollo de software. Es utilizada por una gran variedad de empresas, incluyendo Google, Facebook, Amazon y Microsoft.

Por ejemplo, en una arquitectura serverless, podrías tener una función Lambda en AWS que procese las solicitudes de una API REST. Utilizando OpenAPI, podrías definir y documentar esta API de la siguiente manera:

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Ejemplo de API serverless
```

```yaml
paths:
  /users:
    get:
      summary: Devuelve una lista de usuarios
      responses:
        '200':
          description: Una lista de usuarios
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
```

Esta definición de OpenAPI especifica una API que tiene un punto de acceso GET en "/users" que devuelve una lista de objetos de usuario.

JSON Web Tokens (JWT) es un estándar que define una forma compacta y segura de transmitir información entre partes como un objeto JSON. Esta información puede ser verificada y confiable porque está firmada digitalmente.

Los JWT se utilizan a menudo para la autenticación y la autorización. Pueden utilizarse para transmitir información de identificación, como el nombre de usuario y la contraseña, o información de autorización, como los permisos de acceso a un recurso.

Los JWT se componen de tres partes:

- Cabecera: La cabecera contiene información sobre el JWT, como el tipo de token, el algoritmo de firma y la codificación.
- Carga: La carga contiene la información que se quiere transmitir, como la identificación, los permisos o cualquier otra información.

- Firma: La firma es una huella digital de la cabecera y la carga, que se utiliza para verificar la integridad del token.

Los JWT se pueden firmar con una clave secreta o una clave pública/privada. Si se utiliza una clave secreta, el token solo puede ser verificado por la parte que conoce la clave secreta. Si se utiliza una clave pública/privada, el token puede ser verificado por cualquier parte que tenga la clave pública.

Los JWT tienen una serie de ventajas para la seguridad:

- Integridad: La firma del token garantiza que el token no ha sido modificado.
- Confidencialidad: Si se utiliza una clave secreta, el token solo puede ser leído por la parte que conoce la clave secreta.
- Autenticidad: Si se utiliza una clave pública/privada, el token puede ser verificado por cualquier parte que tenga la clave pública.

Los JWT también tienen una serie de ventajas para la eficiencia:

- Compactos: Los JWT son muy compactos, lo que los hace ideales para el intercambio de datos a través de redes.
- Legibles por máquina: Los JWT están en formato JSON, lo que los hace fáciles de leer y procesar por máquinas.

Por ejemplo, en una arquitectura serverless, podrías tener un sistema de autenticación que use JWT. Cuando un usuario se autentica, el servidor genera un JWT que contiene la identidad del usuario y lo devuelve al usuario. Cuando el usuario realiza solicitudes a la API, incluye este JWT en el encabezado de autorización de la solicitud. La función serverless que procesa la solicitud puede verificar el JWT y usar la identidad del usuario para autorizar la solicitud.

Ejemplo de un JWT:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

En este ejemplo, "sub" es el identificador del usuario, "name" es el nombre del usuario y "iat" es el momento en que se emitió el token.

**Herramientas y servicios de integración**: La aparición de servicios de integración como AWS Lambda, Azure Functions y Google Cloud Functions ha permitido a los desarrolladores crear funciones serverless que se ejecutan en respuesta a eventos específicos, como solicitudes de API.

**Beneficios de la Gestión de API en Arquitecturas Serverless:** Este tipo de soluciones de API Management y herramientas ofrecen numerosos beneficios, entre los cuales se incluyen:

- Escalabilidad: Las arquitecturas serverless escalan automáticamente en función de la demanda, lo que permite a las organizaciones adaptarse rápidamente a las fluctuaciones en el tráfico y los patrones de uso. Esto facilita el manejo de cargas de trabajo variables y picos de tráfico sin la necesidad de provisionar recursos adicionales manualmente.

- Reducción del tiempo de desarrollo: Al desacoplar la lógica de negocio de la gestión de infraestructura, las arquitecturas serverless permiten a los desarrolladores centrarse en la creación de funcionalidades y servicios, reduciendo así el tiempo necesario para desarrollar e implementar nuevas API.

- Mayor seguridad y control: Las plataformas de gestión de API ofrecen una amplia gama de características de seguridad y control, incluida la autenticación, autorización y limitación de la tasa de solicitudes. Esto permite a las organizaciones proteger sus API y datos de accesos no autorizados y abusos, mejorando la confidencialidad e integridad de la información.

Los servicios de gestión de API en arquitecturas serverless son esenciales para el éxito de proyectos modernos de desarrollo de software. Permiten un nivel de abstracción y un grado de control que facilita la implementación y gestión de interfaces de programación de aplicaciones en la nube. Estos servicios ofrecen capacidades de administración de API, como el

enrutamiento y transformación de solicitudes, la limitación de la tasa de solicitudes, la autenticación y la autorización, entre otras funciones.

Los proveedores líderes en la industria de la nube, como Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform (GCP), ofrecen soluciones robustas de administración de API para arquitecturas serverless. Amazon API Gateway, Azure API Management y Google Cloud Endpoints proporcionan funcionalidades avanzadas y son escalables para adaptarse a las necesidades cambiantes de las aplicaciones modernas. Estas alternativas serán analizadas y comparadas en el punto siguiente.

Estos servicios de gestión de API permiten la creación de APIs RESTful y WebSocket, entre otros, que pueden exponer servicios HTTP y AWS Lambda (en el caso de AWS), proporcionan integración con el sistema de autenticación y autorización de cada plataforma y ofrecen monitoreo y rastreo de las solicitudes de API para la detección de problemas y la toma de decisiones basada en datos.

En este ámbito, existen varios tipos de APIs que se pueden generar, cada una con sus propios casos de uso y ventajas. Algunos de los tipos de API más comunes son:

**APIs REST**: Las APIs REST (Representational State Transfer) son un estilo de arquitectura que utiliza protocolos HTTP estándar para realizar operaciones (GET, POST, PUT, DELETE) en recursos representados como URL. REST utiliza un modelo sin estado, lo que significa que cada solicitud del cliente debe contener toda la información necesaria para comprender y procesar la solicitud.

Las APIs REST son un estándar de facto para la comunicación entre aplicaciones web. Son fáciles de implementar y utilizar, y se pueden utilizar para una amplia gama de propósitos, incluyendo el intercambio de datos, la integración de sistemas y la creación de aplicaciones móviles.

Los siguientes son algunos de los conceptos clave de REST:

- Recursos: Los recursos son entidades que son accesibles a través de una API REST. Los recursos pueden ser cualquier cosa, desde datos simples hasta objetos complejos.

- URL: Las URLs se utilizan para identificar recursos. Las URLs deben ser únicas y descriptivas para que sea fácil identificar y acceder a los recursos.
- Operaciones: REST define cuatro operaciones básicas: GET, POST, PUT y DELETE.
- Modelo sin estado: Las APIs REST utilizan un modelo sin estado, lo que significa que no mantienen ninguna información sobre las solicitudes anteriores. Esto hace que las APIs REST sean escalables y eficientes.

Las APIs REST ofrecen una serie de ventajas, entre ellas:

- Facilidad de implementación: Las APIs REST son relativamente fáciles de implementar en comparación con otros estilos de arquitectura.
- Facilidad de uso: Las APIs REST son fáciles de utilizar para los desarrolladores.
- Escalabilidad: Las APIs REST son escalables y pueden soportar un gran volumen de tráfico.
- Eficiencia: Las APIs REST son eficientes y utilizan recursos mínimos.
- Interoperabilidad: Las APIs REST son interoperables con una amplia gama de tecnologías.

Las APIs REST se utilizan en una amplia gama de casos de uso, entre ellos:

- Intercambio de datos: Las APIs REST se pueden utilizar para intercambiar datos entre aplicaciones web.
- Integración de sistemas: Las APIs REST se pueden utilizar para integrar sistemas dispares.
- Creación de aplicaciones móviles: Las APIs REST se pueden utilizar para crear aplicaciones móviles.

Un ejemplo de API REST de pronóstico del tiempo es un caso de uso común. En este caso, cada ciudad es un recurso que se puede acceder a través de una URL como *http://api.weather.com/cities/1234* utilizando el método GET para obtener el pronóstico del tiempo.

Otras operaciones REST que se podrían utilizar en esta API incluyen:

- POST: Para crear una nueva ciudad
- PUT: Para actualizar la información de una ciudad existente

- DELETE: Para eliminar una ciudad

**APIs SOAP** (Simple Object Access Protocol) son un protocolo de intercambio de información estructurada en la implementación de servicios web que utiliza XML. A diferencia de REST, SOAP puede funcionar sobre cualquier protocolo de transporte (HTTP, SMTP, etc.) y ofrece funcionalidades más avanzadas en términos de seguridad y transacciones ACID.

Las APIs SOAP son otro estándar de facto para la comunicación entre aplicaciones empresariales. Son más complejas de implementar y utilizar que las APIs REST, pero ofrecen una serie de ventajas, entre ellas:

- Seguridad: SOAP proporciona un alto nivel de seguridad, gracias a la utilización de XML y a la posibilidad de utilizar protocolos de transporte seguros como HTTPS.
- Transacciones ACID: SOAP permite realizar transacciones ACID, que garantizan que las operaciones se completen de forma consistente o no se completen en absoluto.
- Interoperabilidad: SOAP es interoperable con una amplia gama de tecnologías, incluyendo Java, .NET y Python.

Los siguientes son algunos de los conceptos clave de SOAP:

- XML: SOAP utiliza XML para representar la información que se intercambia entre las aplicaciones.
- Mensajes: SOAP utiliza mensajes para transferir información entre las aplicaciones.
- WSDL: WSDL (Web Services Description Language) es un lenguaje de descripción de servicios web que se utiliza para describir los servicios SOAP.
- UDDI: UDDI (Universal Description, Discovery and Integration) es un directorio de servicios web que se utiliza para encontrar servicios SOAP.

Las APIs SOAP ofrecen una serie de ventajas, entre ellas:

- Seguridad: SOAP proporciona un alto nivel de seguridad, gracias a la utilización de XML y a la posibilidad de utilizar protocolos de transporte seguros como HTTPS.
- Transacciones ACID: SOAP permite realizar transacciones ACID, que garantizan que las operaciones se completen de forma consistente o no se completen en absoluto.
- Interoperabilidad: SOAP es interoperable con una amplia gama de tecnologías.

Las APIs SOAP se utilizan en una amplia gama de casos de uso, entre ellos la integración de sistemas empresariales, el intercambio de datos confidenciales, como información financiera o médica, o la ejecución de transacciones empresariales críticas, como transferencias de dinero o reservas de vuelos.

**APIs GraphQL**: GraphQL es un lenguaje de consulta de datos desarrollado por Facebook que permite a los clientes definir la estructura de los datos requeridos, y la misma estructura de los datos se devuelve desde el servidor, evitando la sobre y subextracción de datos.

Un caso práctico de uso de este tipo de API sería una red social, que podría tener una API GraphQL donde los clientes pueden recuperar exactamente los datos que necesitan, como los nombres y perfiles de los amigos de un usuario, con una sola solicitud, en lugar de realizar varias solicitudes REST.

Ejemplo de llamada API GraphQL para obtener todos los usuarios:

```
query {
  users {
    id
    name
  }
}
```

Esta llamada API devolverá una lista de objetos User con los campos id y name.

Ejemplo de llamada API GraphQL para obtener un usuario específico:

```
query {
  user(id:1) {
    id
    name
  }
}
```

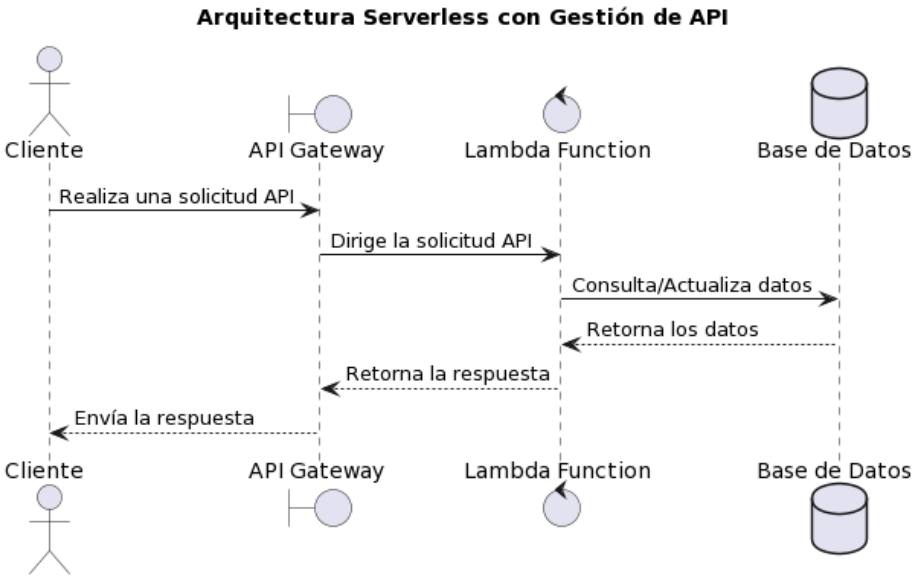
Esta llamada API devolverá un objeto User con los campos id y name para el usuario con el ID 1.

**APIs WebSockets**: WebSocket es un protocolo de comunicación que proporciona canales de comunicación bidireccionales completos sobre una única conexión TCP. Es especialmente útil para aplicaciones en tiempo real. Como ejemplo de este tipo, tendríamos un servicio de chat en tiempo real que podría utilizar una API WebSocket para permitir la comunicación bidireccional en tiempo real entre el cliente y el servidor.

Cada uno de estos tipos de API tiene sus propios usos y beneficios dependiendo del caso de uso. En muchos entornos, puede tener sentido utilizar una combinación de diferentes tipos de API para satisfacer diferentes necesidades.

A medida que las arquitecturas serverless continúan evolucionando, se espera que la gestión de API juegue un papel cada vez más importante en el diseño y desarrollo de aplicaciones y servicios de software. Con la creciente demanda de soluciones de integración más flexibles y escalables, los proveedores de la nube están invirtiendo en la mejora de sus plataformas de gestión de API y en la creación de nuevas herramientas y servicios para facilitar la adopción de arquitecturas serverless.

En particular, se prevé un crecimiento significativo en el uso de servicios de gestión de API para la integración de aplicaciones serverless con servicios de inteligencia artificial y aprendizaje automático, lo que permitirá a las organizaciones desarrollar aplicaciones más inteligentes y capaces de adaptarse a las cambiantes necesidades y comportamientos de los usuarios.




### f. Alternativas de API Management.

El mercado de gestión de API ha experimentado un crecimiento considerable en la última década, impulsado por la creciente necesidad de las organizaciones de compartir funcionalidades y datos de forma segura y eficiente. La gestión de API proporciona una capa esencial en el desarrollo y operación de aplicaciones modernas, permitiendo la interconexión de servicios y datos, tanto interna como externamente[1].

Las API, o interfaces de programación de aplicaciones, son un conjunto de reglas y protocolos que permiten a diferentes softwares interactuar entre sí. La gestión de API se refiere al proceso de creación, publicación, documentación y análisis de estas API. Una solución de gestión de API bien diseñada permitirá a las organizaciones gestionar eficazmente el ciclo de vida completo de una API, desde su creación y despliegue hasta su retirada.

El mercado de gestión de API está dominado por varios proveedores líderes, entre los que se incluyen Amazon Web Services (AWS), Google

1. [1] https://www.gartner.com/reviews/market/full-life-cycle-api-management

Cloud Platform (GCP), Microsoft Azure, IBM, Mulesoft (propiedad de Salesforce), Kong, y Apigee (propiedad de Google). Estos proveedores ofrecen soluciones de gestión de API que varían en funcionalidades, precio y facilidad de uso.

Estas soluciones suelen incluir funcionalidades como portales de desarrolladores, gateways de API, análisis y monitoreo, y seguridad de API. Los portales de desarrolladores permiten a los desarrolladores acceder a documentación, herramientas y soporte para trabajar con las API. Los gateways de API proporcionan un punto de entrada para las solicitudes de API, gestionando la autenticación, la autorización y la transformación de solicitudes. Las funcionalidades de análisis y monitoreo permiten a las organizaciones rastrear el uso de la API y detectar problemas de rendimiento o seguridad.

La elección de una solución de gestión de API depende de una serie de factores, incluyendo las necesidades específicas de la organización, el presupuesto, el lenguaje de programación y la arquitectura de la aplicación, y las capacidades internas de desarrollo y operación.

Varios estudios han propuesto metodologías para comparar y evaluar soluciones de gestión de API. Estas metodologías suelen centrarse en un conjunto de criterios clave, que pueden incluir la funcionalidad, la facilidad de uso, la escalabilidad, la seguridad, y el soporte y la documentación disponibles.

Por ejemplo, la metodología propuesta por Erl (2012) se centra en la funcionalidad de la solución, la facilidad de uso, la escalabilidad y la seguridad. La metodología incluye una serie de pruebas y evaluaciones para cada criterio, que se utilizan para comparar diferentes soluciones de gestión de API.

Otras metodologías se centran en la eficacia de la solución en un contexto específico. Esto incluye la evaluación de la solución en términos de su capacidad para satisfacer las necesidades específicas de la organización, como la gestión de API específicas o la integración con otras herramientas y sistemas.

Estas metodologías proporcionan un marco útil para comparar y evaluar soluciones de gestión de API. Sin embargo, es importante tener en cuenta que la elección de la solución correcta depende en última instancia de las necesidades y capacidades específicas de la organización.

**Soluciones de Gestión de API**

- AWS API Gateway[2]:

AWS API Gateway es un servicio completamente administrado que facilita el desarrollo, la implementación y la administración seguras de APIs en cualquier escala. Proporciona características como la autorización y el control de acceso, el monitoreo del tráfico, la gestión de versiones de API y mucho más.

Una de las principales ventajas de AWS API Gateway es su estrecha integración con otros servicios de AWS, lo que permite a los desarrolladores crear, desplegar y administrar APIs que pueden enrutar solicitudes a varios servicios de AWS, como AWS Lambda, Amazon EC2, Amazon S3 y DynamoDB.

AWS API Gateway es comúnmente utilizado junto con AWS Lambda para crear aplicaciones serverless. En este caso, las funciones de AWS Lambda sirven como backends para manejar las solicitudes de la API. Cuando se invoca una API, AWS API Gateway enruta la solicitud a la función Lambda correspondiente. Esta combinación permite a los desarrolladores centrarse en la lógica de negocio, sin tener que preocuparse por el aprovisionamiento y la gestión de servidores.

AWS API Gateway también se puede utilizar en arquitecturas de microservicios. En este caso, cada microservicio puede exponer su propia API, y AWS API Gateway puede actuar como un único punto de entrada que enruta las solicitudes a los microservicios correspondientes. Esto simplifica la gestión de las APIs y proporciona un mecanismo de descubrimiento de servicios.

[2] https://aws.amazon.com/es/api-gateway/

AWS API Gateway puede actuar como un "escudo" para proteger los servicios de back-end. Puede limitar la tasa de solicitudes para evitar ataques de denegación de servicio, y también puede manejar la autorización y autenticación de las solicitudes utilizando AWS Identity and Access Management (IAM) o Amazon Cognito.

AWS API Gateway ofrece una amplia gama de capacidades que facilitan la construcción, el despliegue y la administración de APIs. A continuación, se detallan algunas de sus características principales:

- Tipos de API: AWS API Gateway soporta la creación de APIs RESTful y WebSocket. Las APIs RESTful son útiles para los casos de uso en los que los clientes interactúan con el back-end mediante solicitudes HTTP estándar. Por otro lado, las APIs WebSocket son útiles cuando se necesita una comunicación bidireccional en tiempo real entre el cliente y el servidor.

- Integración con AWS: API Gateway puede integrarse con otros servicios de AWS para enrutar las solicitudes de la API a los puntos de enlace apropiados. Algunos de los servicios con los que se puede integrar son AWS Lambda, Amazon EC2, Amazon S3, y DynamoDB.

- Gestión del ciclo de vida de las API: AWS API Gateway proporciona herramientas para gestionar todo el ciclo de vida de una API, desde su creación hasta su desactivación. Esto incluye la definición de las API, la gestión de versiones y etapas, la implementación de las API, y la monitorización y mantenimiento de las API en producción.

- Seguridad y autorización: API Gateway ofrece múltiples mecanismos de autorización y seguridad, incluyendo la integración con AWS Identity and Access Management (IAM) y Amazon Cognito para la autenticación de usuarios. También permite la creación de planes de uso para limitar la tasa de solicitudes y el número total de solicitudes que un cliente puede hacer a una API.

- Monitoreo y análisis: API Gateway se integra con Amazon CloudWatch y AWS X-Ray, proporcionando a los desarrolladores

visibilidad sobre el rendimiento de sus APIs y la capacidad de rastrear y solucionar problemas.

- Transformación de solicitudes y respuestas: API Gateway puede transformar las solicitudes y respuestas de la API en tiempo real, permitiendo a los desarrolladores modificar el formato de los datos que se envían y reciben a través de las APIs.

- SDK Generation: API Gateway puede generar SDKs en varios lenguajes (como JavaScript, iOS y Android) para ayudar a los desarrolladores a construir aplicaciones que consumen las APIs.

- Despliegue y gestión de APIs privadas: API Gateway permite a los desarrolladores crear APIs privadas que solo pueden ser accedidas desde dentro de una red de Amazon VPC, proporcionando una capa adicional de seguridad

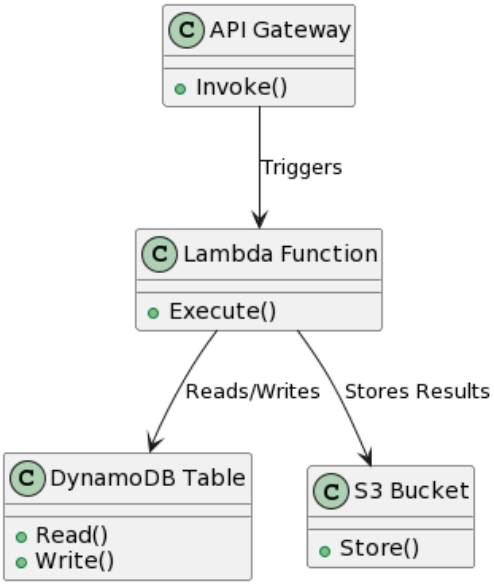



Ilustración 6 Ejemplo básico de uso de AWS API Gateway conectado con otros servicios de AWS

- Azure API Management[3]

[3] https://azure.microsoft.com/en-us/products/api-management

Azure API Management es otra solución integral que permite a las organizaciones publicar, gestionar, proteger y analizar sus APIs de manera eficaz. Al igual que otras soluciones de gestión de APIs, proporciona una serie de características y capacidades que facilitan la implementación y el mantenimiento de APIs.

Entre los principales casos de uso que podemos encontrar específicos para esta solución tenemos:

- Creación de APIs unificadas: Azure API Management permite a las organizaciones unificar sus APIs independientemente de dónde estén alojadas, ya sea en la nube, en local o en una combinación de ambas. Esto facilita a los desarrolladores la interacción con las APIs y les proporciona una visión unificada.

- Exposición de servicios como APIs: Las empresas pueden utilizar Azure API Management para exponer sus servicios como APIs, lo que facilita su consumo y integración con otras aplicaciones y servicios.

- Monetización de APIs: Azure API Management permite a las organizaciones monetizar sus APIs mediante la creación de planes de suscripción y la aplicación de cuotas y tarifas.

- Control de acceso y seguridad: Las empresas pueden utilizar Azure API Management para implementar políticas de control de acceso y seguridad, protegiendo sus APIs de usos no autorizados o malintencionados.

Las capacidades de una solución de gestión de API son un conjunto crucial de características que determinan su eficacia en la administración de las APIs de una organización. Estas capacidades pueden abarcar desde la gestión del ciclo de vida de las API, la seguridad y la protección, hasta la transformación de solicitudes y respuestas, la integración con otras plataformas y la monitorización y análisis.

A continuación, se presentan las capacidades de Azure API Management, que se destacan por su exhaustividad y flexibilidad, y que ayudan a las organizaciones a maximizar el valor de sus APIs:

- Gestión del ciclo de vida de las API: Azure API Management proporciona herramientas para la creación, publicación, mantenimiento, versión y deprecación de APIs.

- Seguridad y protección: Azure API Management soporta una variedad de esquemas de autenticación y autorización, incluyendo OAuth 2.0 y OpenID Connect. También proporciona protección contra amenazas comunes como los ataques DDoS.

- Transformación de solicitudes y respuestas: Al igual que AWS API Gateway, Azure API Management permite transformar las solicitudes y respuestas de la API, lo que puede ser útil para cambiar el formato de los datos o para modificar las respuestas antes de enviarlas al cliente.

- Integración con Azure y otras plataformas: Azure API Management puede integrarse con otros servicios de Azure, así como con otras plataformas y servicios, lo que facilita la creación de soluciones complejas.

- Monitorización y análisis: Azure API Management se integra con Azure Monitor y Azure Application Insights, proporcionando a los desarrolladores visibilidad sobre el rendimiento y la utilización de sus APIs.

- Portal de desarrolladores: Azure API Management incluye un portal de desarrolladores auto-servicio donde los desarrolladores pueden aprender sobre las APIs, probarlas y suscribirse a ellas.

- Políticas de API flexibles: Azure API Management permite a los administradores aplicar políticas de API para controlar y modificar el comportamiento de las API.

- Google Cloud Endpoints[4]

Google Cloud Endpoints es una solución de gestión de API de Google Cloud que facilita el desarrollo, la implementación, la protección y el monitoreo de las APIs en Google Cloud. Con la capacidad de admitir APIs HTTP/1.1 y HTTP/2, Endpoints se integra perfectamente con Google Cloud y otros servicios de Google, lo que facilita la creación de aplicaciones y servicios escalables y seguros.

Endpoints es particularmente útil en casos de uso como:

- Desarrollo de aplicaciones móviles y web: Cloud Endpoints proporciona las herramientas necesarias para desarrollar, implementar y gestionar APIs que se utilizan en aplicaciones móviles y web. Proporciona autenticación y autorización incorporadas, lo que permite controlar quién tiene acceso a la API y qué acciones puede realizar.

- Implementación de microservicios: Cloud Endpoints puede ser utilizado para implementar y gestionar APIs para microservicios. La capacidad de escalar automáticamente y el monitoreo incorporado hacen que sea una opción ideal para la gestión de microservicios en Google Cloud.

- Desarrollo de aplicaciones de IoT: La solución de gestión de API de Google puede ser utilizada para desarrollar y gestionar APIs para dispositivos IoT. La capacidad de manejar un gran número de conexiones simultáneas y el monitoreo en tiempo real hacen de Endpoints una opción atractiva para las aplicaciones de IoT.

[4] https://cloud.google.com/endpoints?hl=es

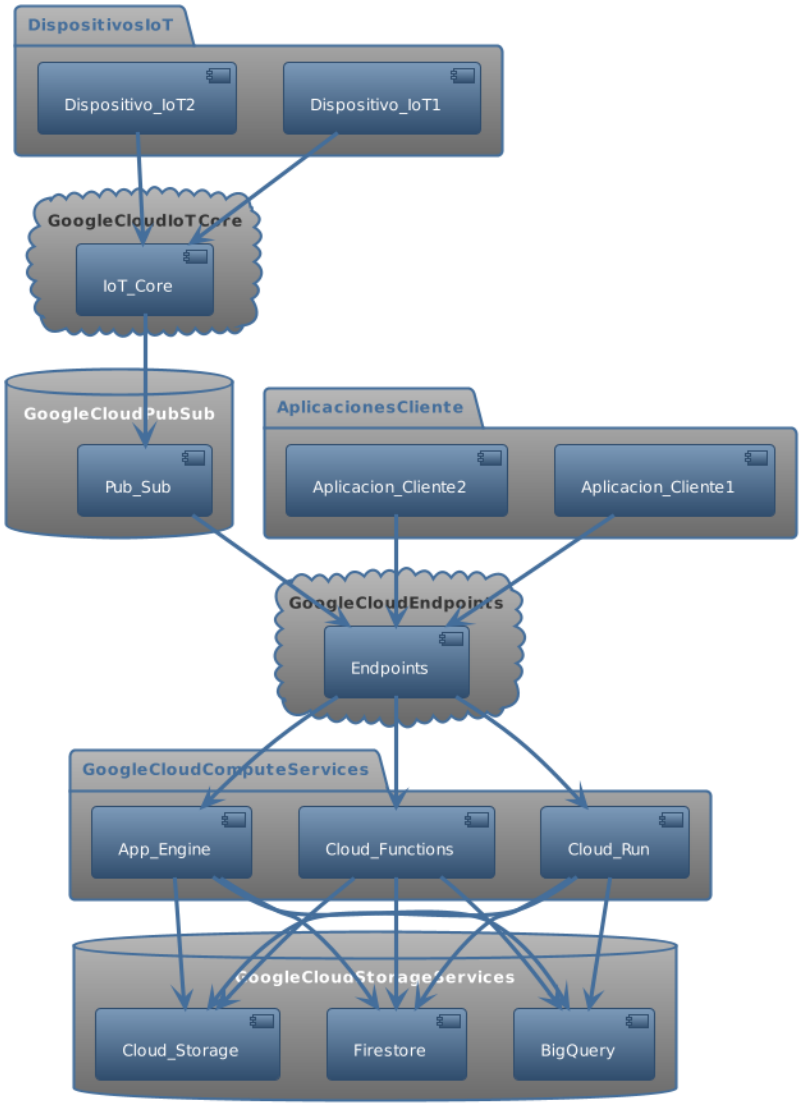



Ilustración 7 Diagrama de una arquitectura competa de IoT sobre Google Cloud

Las capacidades de Google Cloud Endpoints incluyen:

- Gestión del ciclo de vida de las API: Google Cloud Endpoints ofrece control completo sobre el ciclo de vida de las API, desde la creación hasta la desactivación.

- Autenticación y autorización: Cloud Endpoints utiliza Firebase Authentication, Auth0 y Google Sign-In para autenticar a los usuarios de la API.

- Escalabilidad y rendimiento: Cloud Endpoints se escala automáticamente para manejar las solicitudes a la API, independientemente del tráfico.

- Monitoreo y logging: Cloud Endpoints se integra con Stackdriver Monitoring y Logging, proporcionando visibilidad en tiempo real sobre el uso y el rendimiento de la API.

- Soporte para el protocolo OpenAPI: Cloud Endpoints soporta el protocolo OpenAPI, lo que facilita la interoperabilidad con otras herramientas y servicios.

- Integración con Google Cloud: Cloud Endpoints se integra perfectamente con otros servicios de Google Cloud, incluyendo Google Kubernetes Engine (GKE), App Engine y Compute Engine.

- Mulesoft[5]:

Mulesoft's Anypoint Platform™ (MuleSoft), una empresa adquirida por Salesforce en 2018, ofrece una solución de gestión de API llamada Anypoint Platform. Esta plataforma proporciona una amplia gama de características para el diseño, la construcción, la implementación, la gestión y la seguridad de las API y las integraciones.

Las capacidades diferenciales de MuleSoft Anypoint Platform, en comparación con otros productos de gestión de API, pueden incluir:

- Integración de sistemas heredados: MuleSoft se destaca en la integración de sistemas heredados con tecnologías más nuevas. Esto permite a las empresas preservar y aprovechar sus inversiones existentes mientras avanzan hacia tecnologías más modernas.
- Conectividad omnicanal: MuleSoft proporciona capacidades sólidas para integrar datos y servicios en múltiples canales, lo que ayuda a las

[5] https://www.mulesoft.com/platform/api-management

empresas a ofrecer experiencias de usuario consistentes a través de todos los puntos de contacto.

- Unificación de la integración de aplicaciones en la nube y en local: MuleSoft ofrece una plataforma unificada para integrar aplicaciones tanto en la nube como en local, lo que proporciona flexibilidad a las empresas que desean moverse hacia la nube a su propio ritmo.
- Gestión completa del ciclo de vida de las API: MuleSoft proporciona una gama completa de herramientas para gestionar todo el ciclo de vida de las API, desde el diseño hasta la retirada. Esto incluye capacidades para el diseño, la construcción, la implementación, la gestión, la seguridad, el monitoreo y el análisis de las API.
- Amplias capacidades de personalización y extensión: A través del lenguaje de expresión DataWeave y el lenguaje de scripting Mule, los desarrolladores pueden personalizar y extender la funcionalidad de la plataforma para satisfacer necesidades específicas.
- Integración con Salesforce: Como parte de la familia Salesforce, MuleSoft se integra bien con otras soluciones de Salesforce, lo que puede ser un diferenciador significativo para las empresas que ya utilizan o planean utilizar productos de Salesforce.
- Comunidad activa y extensa documentación: MuleSoft tiene una comunidad de desarrolladores activa y proporciona documentación detallada y completa, lo que puede facilitar la adopción y el uso de la plataforma.

- Kong[6]

Kong es una solución de gestión de API de código abierto y centrada en la nube que proporciona funcionalidades de alto rendimiento para el enrutamiento, la conectividad, la lógica de negocio y la seguridad de las API. Kong es conocido por su flexibilidad, escalabilidad y fácil adaptabilidad a diferentes infraestructuras, desde las monolíticas hasta las basadas en microservicios y serverless.

Las capacidades diferenciales de Kong que lo distinguen de otras soluciones pueden incluir:

- Código abierto y extensibilidad: A diferencia de muchas soluciones de gestión de API, Kong es una plataforma de código abierto. Esto permite a los desarrolladores personalizar y ampliar la plataforma de

[6] https://konghq.com/products/kong-gateway

acuerdo con sus necesidades específicas. Además, Kong proporciona una amplia gama de plugins para extender la funcionalidad de la plataforma.

- Rendimiento y escalabilidad: Kong es conocido por su alto rendimiento y escalabilidad, lo que lo hace adecuado para empresas que manejan un gran volumen de tráfico de API.

- Soporte para múltiples protocolos: Además de HTTP y HTTPS, Kong también soporta protocolos como gRPC, TCP, TLS, UDP, y más. Esta flexibilidad en el soporte de protocolos puede ser útil en escenarios de uso complejos.

- Servicio Mesh con Kuma: Kong ofrece la capacidad de utilizar Kuma, un controlador de servicio mesh de código abierto, lo que permite a los desarrolladores controlar el tráfico de red y asegurar las comunicaciones entre los servicios a un nivel muy granular.

- Interoperabilidad con Kubernetes: Kong tiene una integración profunda con Kubernetes, lo que lo hace una elección atractiva para las organizaciones que utilizan este sistema de orquestación de contenedores.

- Seguridad avanzada: Kong ofrece capacidades de seguridad avanzadas, incluyendo la autenticación y autorización, la limitación de la tasa, el control de acceso basado en roles (RBAC) y mucho más.

- Amplia comunidad y soporte: Siendo una plataforma de código abierto, Kong cuenta con una amplia comunidad de desarrolladores que contribuyen a su evolución y mejoramiento. Además, ofrece soporte empresarial para las organizaciones que necesitan garantías adicionales.

Tabla comparativa de servicios serverless de gestión de API

| Característica | AWS API Gateway | Azure API Management | GCP Cloud Endpoints | MuleSoft | Kong |
|---|---|---|---|---|---|
| **Creación y despliegue de API** | Sí | Sí | Sí | Sí (A través de Anypoint Platform) | Sí |
| **Soporte para REST y WebSocket** | Sí (REST), Sí (WebSocket) | Sí (REST), No (WebSocket) | Sí (REST), No (WebSocket) | Sí | Sí |
| **Importar y exportar definiciones de API** | Sí (Con OpenAPI) | Sí (Con OpenAPI) | Sí (Con OpenAPI y gRPC API) | Sí (Con RAML y OAS) | Sí (Con OpenAPI) |
| **Autenticación y autorización** | Sí (Con Cognito) | Sí (Con Azure AD) | Sí (Con Google Cloud IAM) | Sí (Con Anypoint Security) | Sí (Con plugins de autenticación) |
| **Limitación de solicitudes (Throttling)** | Sí | Sí | Sí | Sí (A través de políticas) | Sí (Con plugins de limitación) |
| **Monitoreo y registro** | Sí (Con CloudWatch) | Sí (Con Azure Monitor) | Sí (Con Stackdriver) | Sí (Con Anypoint Monitoring) | Sí (A través de la interfaz de Kong) |
| **Caché** | Sí | Sí | No | Sí (A través de políticas) | Sí (Con plugins de caché) |
| **Transformación de solicitudes y respuestas** | Sí | Sí | Sí | Sí (Con DataWeave) | Sí (Con plugins de transformación) |
| **Pruebas e implementaciones de etapas** | Sí | Sí | Sí | Sí (Con Anypoint Testing) | Sí (Con plugins de prueba) |
| **Integración con otros servicios** | Sí (Con servicios de AWS) | Sí (Con servicios de Azure) | Sí (Con servicios de Google Cloud) | Sí (Con Anypoint Exchange) | Sí (Con plugins de integración) |

**Comparación de precios**

El costo es un factor crítico para tener en cuenta al seleccionar una solución de gestión de API. En este punto realizaremos una comparación de los planes de precios de AWS API Gateway, Azure API Management y GCP Cloud Endpoints, así como de Mulesoft y Kong:

- AWS API Gateway: AWS API Gateway utiliza un modelo de precios basado en el uso, en el que se paga por las llamadas a la API y el traslado de datos. El precio varía según el tipo de endpoint (público, privado o de interfaz). AWS también ofrece una capa gratuita que incluye un millón de llamadas a la API por mes durante los primeros 12 meses. Después de la capa gratuita, se aplican tarifas según la región y la cantidad de llamadas a la API y los datos transferidos.

- Azure API Management: Azure API Management ofrece varios niveles de precios que varían en características y rendimiento. Los niveles van desde el nivel de desarrollador, que es para pruebas y desarrollo y no está destinado a cargas de trabajo de producción, hasta el nivel premium, que ofrece más capacidad y características adicionales, como VPN, multi-región y Azure AD. También hay una capa de consumo que es de pago por uso y adecuada para cargas de trabajo serverless.

- GCP Cloud Endpoints: Cloud Endpoints cobra en función del número de llamadas a la API. No hay un costo adicional para el uso de Endpoints Frameworks o el proxy Extensible Service Proxy (ESP), pero sí puede haber costos asociados con otros servicios de Google Cloud utilizados con Endpoints, como App Engine, Compute Engine y el almacenamiento de logs en Stackdriver.

- Mulesoft: Mulesoft no proporciona información de precios directamente en su sitio web, pero ofrece varios niveles de servicios. Los precios generalmente se basan en una combinación de características necesarias, como el número de aplicaciones, la cantidad de transacciones y otros factores. Para obtener detalles

específicos, Mulesoft requiere que te pongas en contacto con ellos directamente.

- Kong: Kong ofrece tanto una versión de código abierto gratuita como planes de precios empresariales. Kong Enterprise incluye características avanzadas y soporte y se cobra anualmente basándose en la cantidad de nodos que se utilizan. Para obtener detalles de precios específicos, requiere ponerse en contacto con ellos directamente.

Para terminar de comparar las soluciones de gestión de API de forma exhaustiva, es importante enumerar sus ventajas y desventajas potenciales.

AWS API Gateway

- Ventajas:
  - Integración nativa con el ecosistema AWS.
  - Soporte para WebSocket además de HTTP.
  - Control de acceso basado en IAM de AWS y Cognito.
  - Soporte para la generación de SDK en varios lenguajes.
- Desventajas:
  - Su rendimiento puede verse afectado por la latencia en las funciones Lambda.
  - Puede resultar costoso para aplicaciones con un tráfico de API muy elevado.

Azure API Management

- Ventajas:
  - Integración nativa con servicios Azure.
  - Tiene una política muy robusta y un sistema de reglas.
  - Soporte para OpenAPI y transformaciones de API mediante políticas.
- Desventajas:
  - La configuración y el despliegue pueden ser más complejos en comparación con otras opciones.
  - Su rendimiento puede verse afectado por la latencia de Azure Functions.

Google Cloud Endpoints

- Ventajas:
  - Integración nativa con Google Cloud Platform.
  - Monitoreo y logging proporcionados por Stackdriver.
  - Capacidad para manejar un gran número de conexiones simultáneas.
- Desventajas:
  - Soporte más limitado para protocolos en comparación con otras soluciones.
  - El coste puede aumentar significativamente con el volumen de uso.

Mulesoft API Management

- Ventajas:
  - Proporciona una solución completa para la gestión de APIs, incluyendo diseño, implementación, seguridad y análisis.
  - Permite implementar APIs tanto en la nube como en entornos locales.
  - Ofrece integraciones de preconstrucción para varios servicios y aplicaciones comunes.
- Desventajas:
  - Es una de las soluciones más caras del mercado.
  - La curva de aprendizaje puede ser más alta debido a su amplio conjunto de funcionalidades.

Kong

- Ventajas:
  - Es open-source y por lo tanto más flexible y personalizable.
  - Se puede desplegar en cualquier lugar: en la nube, en el entorno local, o en un entorno híbrido.
  - Soporta una amplia gama de plugins para extender su funcionalidad.
- Desventajas:
  - Requiere más configuración y gestión que las soluciones de gestión de APIs basadas en la nube.
  - La versión de empresa, que incluye características adicionales y soporte, puede resultar cara.

## 4. Transformación de una aplicación monolítica: un caso práctico.

### a. Contexto

Una reconocida compañía de venta de muebles de diseño online ha experimentado un rápido crecimiento en los últimos años. La empresa comenzó con un enfoque de comercio electrónico tradicional, vendiendo sus productos a través de una aplicación monolítica en la web. La aplicación maneja funciones críticas como la autenticación de usuarios, la gestión de pedidos, los pagos y el inventario de productos.

Sin embargo, ha experimentado desafíos relacionados con la escalabilidad y la eficiencia debido a su estructura monolítica. El equipo de TI ha contrastado que pequeñas modificaciones en un módulo de la aplicación requieren desplegar toda la aplicación nuevamente, lo que ha resultado en tiempos de inactividad no deseados y ha ralentizado la velocidad de lanzamiento de nuevas características.

Además, debido al aumento en el número de usuarios y pedidos, los costos de infraestructura están aumentando rápidamente y la empresa ha identificado la necesidad de optimizar estos costos.

Se planea expandirse a nuevos mercados globales en el próximo año. Para soportar esta expansión, la compañía necesita que su sistema sea altamente escalable y resiliente para manejar el aumento esperado en el tráfico web y las transacciones.

La empresa ha asignado un importante presupuesto para su transformación digital, con el objetivo de mejorar la eficiencia operativa, optimizar los costos de infraestructura y mejorar la experiencia del cliente. Como parte de este plan, está considerando la transformación de su aplicación monolítica a una arquitectura serverless.

La arquitectura serverless proporcionará a la empresa la flexibilidad necesaria para escalar automáticamente según la demanda, lo que significa que solo pagarán por los recursos que realmente usen. Además, al descomponer la aplicación en microservicios, podrán realizar actualizaciones y cambios más frecuentes y con menos riesgo, mejorando la velocidad y la eficiencia de sus procesos de desarrollo.

Este contexto de empresa es un escenario común para muchas organizaciones que buscan modernizar sus aplicaciones para ser más competitivas en el mundo digital actual. La transformación de una arquitectura monolítica a serverless puede ofrecer beneficios significativos en términos de eficiencia, escalabilidad y costo.

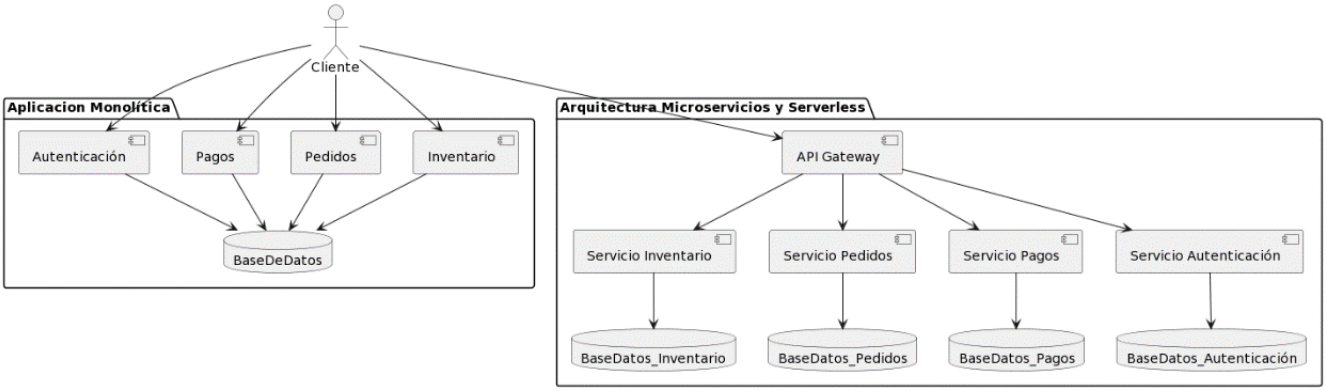


Ilustración 8. Esquemas aplicación monolítica vs arquitectura de microservicios y serverless

### b. Análisis y Planificación

**Análisis del Sistema Monolítico**: En este punto se debe examinar a fondo la aplicación monolítica existente para comprender su funcionalidad y cómo está estructurada.

En el caso práctico tenemos una aplicación monolítica de comercio electrónico con funciones de autenticación, pedidos, pagos e inventario de productos.

El análisis funcional de la misma nos revela los siguientes módulos:

- Funciones de Autenticación: Este módulo maneja el registro de usuarios, el inicio de sesión y la gestión de roles de usuario. El sistema de autenticación puede incluir características como el restablecimiento de contraseñas, la verificación de correos electrónicos y la autenticación de dos factores.

- Pedidos: Este módulo maneja la creación y gestión de pedidos por parte de los usuarios. Podría incluir características como el seguimiento de pedidos, la actualización del estado de los pedidos y la gestión de devoluciones.

- Pagos: Este módulo maneja las transacciones de pago. Puede incluir características como la selección de la opción de pago, la integración con proveedores de servicios de pago y la gestión de reembolsos.

- Inventario de Productos: Este módulo maneja el listado de productos y la gestión de inventario. Puede incluir características como la adición de nuevos productos, la actualización de detalles de productos y la gestión de stock.

Al entender la estructura y funcionalidad de la aplicación, podemos identificar cuáles son los módulos que pueden dividirse en servicios independientes en la arquitectura serverless. Para el próximo paso, es esencial tener un diagrama o esquema de la aplicación para visualizar las dependencias y las interacciones entre estos módulos. De esta manera, podemos planificar cómo se separarán y cómo interactuarán una vez que se haya realizado la transformación.

Este análisis inicial es crítico para garantizar una transición exitosa hacia una arquitectura serverless. Una comprensión sólida de la aplicación monolítica y de cómo sus componentes interactúan entre sí es esencial para

identificar posibles desafíos, prever complicaciones y planificar una estrategia efectiva para la transformación.

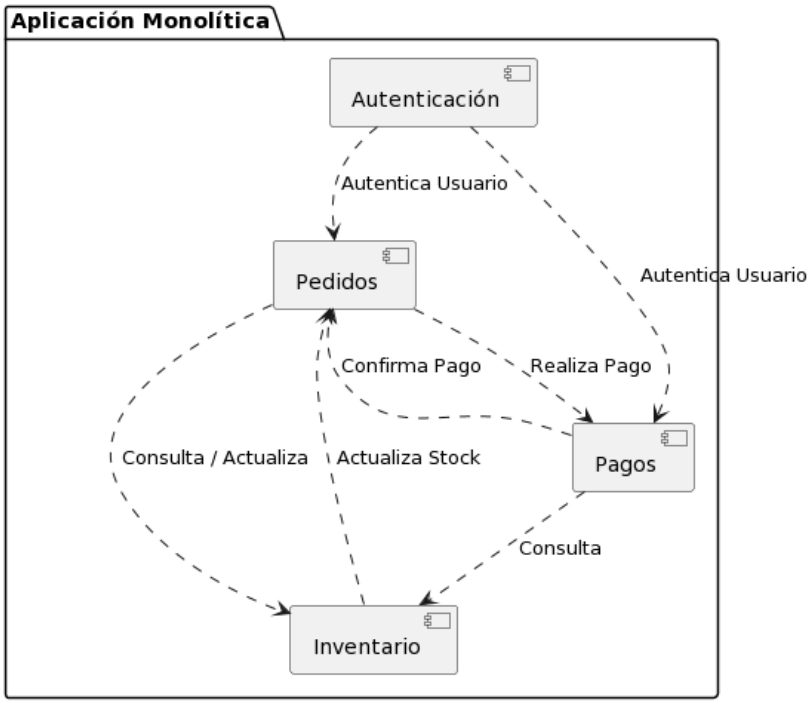



Ilustración 9 Relación funcional entre módulos de la aplicación monolítica

Este diagrama muestra que existen 4 áreas en la aplicación actual:

- ✓ El módulo de Autenticación es el responsable de autenticar a los usuarios que interactúan con los módulos de Pedidos y Pagos.
- ✓ El módulo de Pedidos interactúa con los módulos de Pagos e Inventario para gestionar los pedidos de los clientes.
- ✓ El módulo de Pagos interactúa con el módulo de Pedidos para confirmar los pagos y consulta el Inventario.
- ✓ Finalmente, el módulo de Inventario es consultado y actualizado por los módulos de Pedidos y Pagos para mantener un registro actualizado del stock de productos.

El análisis de estos componentes y sus interacciones nos ayudará a identificar las funciones que podrían convertirse en microservicios en el proceso de transformación hacia una arquitectura serverless.

**Identificar Módulos:** Identificaremos qué partes de la aplicación monolítica se pueden dividir en módulos independientes.

En base al análisis de la aplicación monolítica y el diagrama funcional, hemos identificado los siguientes módulos que se pueden convertir en microservicios independientes:

- Autenticación: Este módulo es responsable de autenticar a los usuarios que utilizan la plataforma. Su transformación en un microservicio independiente permitirá que otras partes de la aplicación verifiquen la identidad del usuario de manera segura y eficiente.

- Pedidos: Este módulo maneja la creación, modificación y eliminación de pedidos realizados por los usuarios. Como microservicio, podrá escalar para manejar grandes volúmenes de pedidos e interactuar con los microservicios de Pagos e Inventario.

- Pagos: Este módulo se encarga de procesar los pagos de los pedidos. Su transformación en un microservicio permitirá integrar diferentes pasarelas de pago y manejar la lógica de pago de manera independiente.

- Inventario: Este módulo es responsable de mantener y actualizar la lista de productos disponibles. Al convertirse en un microservicio, puede escalar para manejar un gran número de productos y solicitudes de actualización de inventario.

**Planificar la Transformación**: Se creará un plan detallado de cómo se realizará la transformación, incluyendo un mapa de cómo se conectarán los diferentes módulos.

Para realizar la transformación de la aplicación monolítica a una arquitectura de microservicios, se propone el siguiente plan:

1. Transformación del Módulo de Autenticación: Este sería el primer módulo por transformar debido a su naturaleza crítica y a su uso en toda la aplicación. Este microservicio deberá proveer una API que permita a otros microservicios verificar la autenticidad del usuario.

2. Transformación del Módulo de Inventario: Este módulo se puede transformar a continuación, ya que tiene interacciones limitadas con otros módulos. Su transformación permitirá el manejo eficiente del inventario.

3. Transformación del Módulo de Pedidos: Una vez que los módulos de Autenticación e Inventario estén transformados, el Módulo de Pedidos puede ser dividido. Este microservicio necesitará interactuar con ambos microservicios para funcionar.

4. Transformación del Módulo de Pagos: Por último, el Módulo de Pagos se puede transformar. Este microservicio interactuará con el Módulo de Pedidos y, posiblemente, con el Módulo de Inventario para confirmar que los productos pedidos están disponibles.

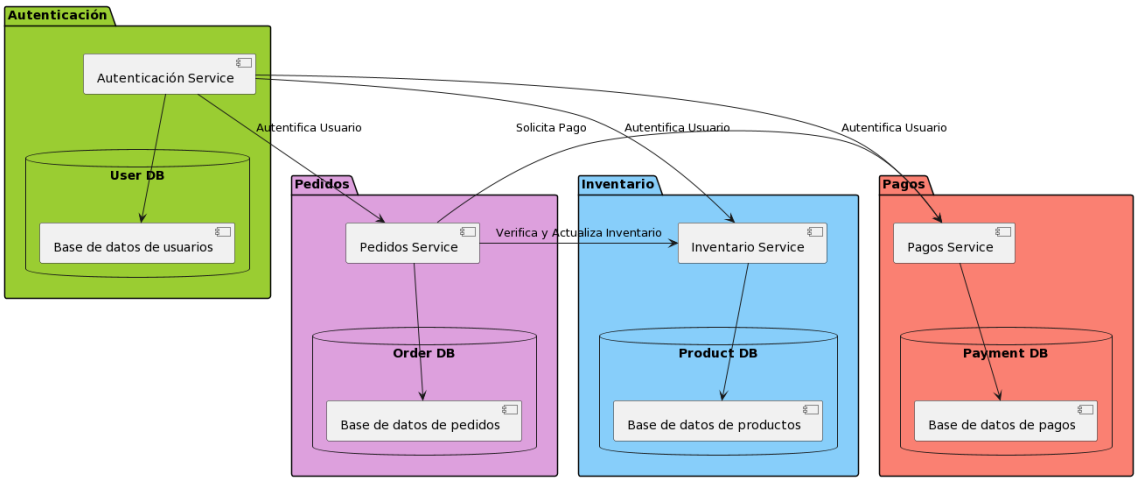



Ilustración 10 Aplicaciones microservicios relacionadas

A lo largo de este proceso, es importante tener en cuenta que cada microservicio debe ser diseñado para ser autónomo e independiente, de tal manera que una falla en un microservicio no afecte al funcionamiento de los demás.

Cada microservicio se desplegará utilizando la arquitectura serverless, lo que permitirá una escalabilidad eficiente y reducirá los costos operativos.

A continuación, presentamos un diagrama de Gantt que proporciona una visión detallada de la planificación temporal para la transformación de una aplicación monolítica a una arquitectura basada en microservicios. Este proceso se ha dividido en varios proyectos, cada uno correspondiente a la creación de un microservicio específico: Autenticación, Inventario, Pagos y Pedidos.

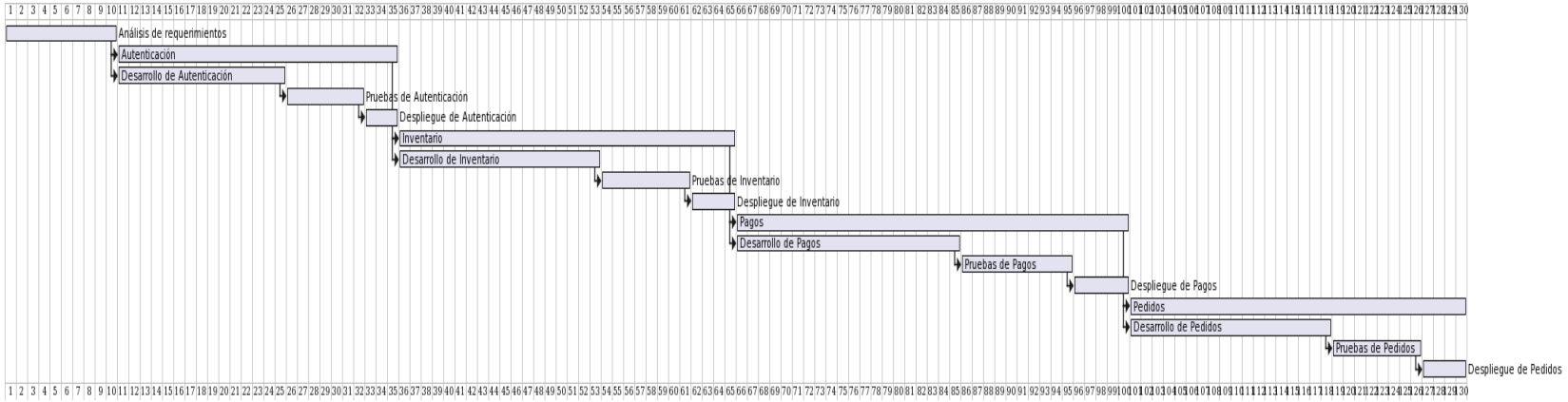
Cada proyecto se ha desglosado en tres fases principales: Desarrollo, Pruebas y Despliegue. En el Desarrollo, los equipos trabajan en la construcción del microservicio basándose en las especificaciones y los requisitos identificados durante el análisis inicial.

A continuación, el microservicio entra en la fase de Pruebas, donde se comprueba su correcto funcionamiento y se identifican y corrigen los posibles problemas. Finalmente, en la fase de Despliegue, el microservicio se pone en producción y se integra con el resto de la arquitectura.

Es importante mencionar que la duración de cada fase se ha ajustado teniendo en cuenta la complejidad inherente a cada microservicio. Por ejemplo, se ha asignado más tiempo a la fase de desarrollo del microservicio de Pagos, dada la importancia de garantizar la seguridad y el cumplimiento de las normativas en todas las transacciones.

El diagrama de Gantt permite visualizar claramente cómo se suceden los proyectos, proporcionando una estimación de la duración total del proceso de transformación. Sin embargo, es importante recordar que estos tiempos son estimaciones y podrían variar en función de una serie de factores, como los

imprevistos que puedan surgir durante el desarrollo o las revisiones necesarias tras las pruebas.


Análisis de requerimientos
Autenticación
Desarrollo de Autenticación
Pruebas de Autenticación
Despliegue de Autenticación
Inventario
Desarrollo de Inventario
Pruebas de Inventario
Despliegue de Inventario
Pagos
Desarrollo de Pagos
Pruebas de Pagos
Despliegue de Pagos
Pedidos
Desarrollo de Pedidos
Pruebas de Pedidos
Despliegue de Pedidos

### c. Desacoplamiento de la Aplicación Monolítica

En el proceso de transformación de una aplicación monolítica a una arquitectura basada en microservicios, un paso crítico es el desacoplamiento de la aplicación monolítica. Este proceso implica la descomposición de la aplicación en módulos o servicios independientes que pueden operar y evolucionar de forma autónoma.

El desacoplamiento de una aplicación monolítica ofrece varias ventajas, incluyendo mayor flexibilidad, escalabilidad y resiliencia. Cada módulo puede ser desarrollado, desplegado y escalado de forma independiente, lo que permite a las organizaciones responder más rápidamente a las necesidades cambiantes del negocio.

A continuación, describiremos los pasos principales en este proceso de desacoplamiento, centrando nuestra atención en la utilización de los servicios de AWS, especialmente las funciones Lambda y la infraestructura serverless.

**Extraer Módulos**

El primer paso en el desacoplamiento de la aplicación monolítica es identificar y extraer módulos de la aplicación. Un módulo es una unidad funcional independiente que realiza una función específica dentro de la aplicación.

Para nuestro caso, hemos identificado cuatro módulos principales: Autenticación, Inventario, Pagos y Pedidos. Cada uno de estos módulos representa una parte crítica de la funcionalidad de la aplicación y será convertido en un servicio independiente.

Para realizar la extracción de los módulos, se utilizarán las funciones AWS Lambda. AWS Lambda permite ejecutar código sin tener que aprovisionar o administrar servidores. Esto proporciona una gran flexibilidad para desarrollar

y desplegar cada módulo como una función independiente que puede ser invocada a través de eventos específicos.

El lenguaje de programación que utilizaremos para el desarrollo de estas funciones es Node.js. Este lenguaje es ampliamente utilizado para el desarrollo de aplicaciones serverless debido a su eficiencia y rendimiento en aplicaciones basadas en I/O, además de su compatibilidad con JSON y su gran ecosistema de módulos.

En la extracción de módulos, es importante tener en cuenta las dependencias entre ellos. Idealmente, cada módulo debe ser lo más independiente posible de los demás. Sin embargo, en la práctica, puede haber dependencias que necesitan ser gestionadas cuidadosamente durante el proceso de extracción para evitar problemas de funcionamiento.

Se proporciona un ejemplo sencillo de cómo se implementar el módulo de inventario con AWS Lambda en Node.js y una base de datos Aurora Serverless.

En este ejemplo, se crea una función Lambda que se conecta a la base de datos y puede realizar tres operaciones: obtener un producto, actualizar el precio de un producto y modificar la cantidad disponible de un producto. Para realizar estas operaciones, utilizaremos el módulo *`mysql2/promise`* para interactuar con la base de datos MySQL de AWS Aurora Serverless.

```
const mysql = require('mysql2/promise');

exports.handler = async (event) => {
    const pool = await mysql.createPool({
        host     : process.env.DB_HOST,
        user     : process.env.DB_USER,
        password : process.env.DB_PASS,
        database : process.env.DB_NAME
    });

    let result;

    switch(event.action) {
        case 'getProduct':
            [result] = await pool.query(`SELECT * FROM products WHERE id = ?`,
```

```
[event.productId]);
            break;

        case 'updatePrice':
            [result] = await pool.query(`UPDATE products SET price = ? WHERE id = ?`,
[event.newPrice, event.productId]);
            break;

        case 'updateQuantity':
            [result] = await pool.query(`UPDATE products SET quantity = ? WHERE id = ?`,
[event.newQuantity, event.productId]);
            break;

        default:
            result = { message: 'Invalid action' };
            break;
    }

    await pool.end();

    return result;
};
```

Para cada acción que desees realizar, necesitarás enviar un objeto event con la acción requerida y los datos necesarios para esa acción. Por ejemplo, para actualizar el precio de un producto, se podría enviar el siguiente objeto event:

```
{
    "action": "updatePrice",
    "productId": "123",
    "newPrice": "99.99"
}
```

### Crear APIs

En este punto hemos de desarrollar APIs para cada módulo de modo que puedan interactuar entre sí. Estas APIs son vitales en la arquitectura de microservicios, ya que cada servicio necesita comunicarse con otros para funcionar correctamente. Por lo tanto, es crucial desarrollar APIs robustas, escalables y seguras.

Hemos decido implementarlo sobre AWS, con una de las plataformas más potentes para el desarrollo y gestión de APIs: AWS API Gateway. AWS API Gateway es un servicio totalmente administrado que facilita a los desarrolladores la creación, publicación, mantenimiento, monitoreo y protección de APIs a cualquier escala. Se pueden crear APIs que accedan a AWS o a otros servicios web, así como a datos almacenados en la nube de AWS.

A continuación, se muestra cómo crear una API con AWS API Gateway para el módulo de inventario que desarrollamos anteriormente con AWS Lambda:

- Acceso a la consola de API Gateway: Se inicia sesión en AWS y se navega hasta la consola de API Gateway. Desde este punto se pueden ver todas las APIs existentes y crear nuevas.

- Creación de una nueva API REST: Desde la opción 'Crear API', se selecciona el tipo 'API REST'. Se proporciona un nombre para la API: "InventarioAPI", y se procede a su creación.

- Configurar un nuevo recurso y su método: Desde la opción 'Acciones' se selecciona 'Crear recurso'. Un recurso representa una entidad que puede ser accedida a través del API, como un producto en el inventario. Después de crear un recurso, se ha de configurar un método para ese recurso, como GET, POST, PUT o DELETE.

- Integrar la API con la función Lambda: Para conectar la API con la función Lambda de inventario, se necesita configurar la integración de Lambda para el método que se ha creado anteriormente. Se seleccionad la función Lambda y guardan los cambios.

- Implementar la API: Finalmente, para que la API esté disponible, se debe desplegar. Desde la opción de 'Acciones' se selecciona 'Implementar API'. Se selecciona un nuevo escenario de implementación y se procede a 'Implementar'.

En este punto, la API está lista y puede ser utilizada para interactuar con la función Lambda de inventario.

Para desarrollar APIs de manera más efectiva y consistente, puede ser útil utilizar un lenguaje de descripción de API, como Swagger o OpenAPI. Estos lenguajes te permiten diseñar un API de antemano y generar documentación automáticamente, lo que facilita la colaboración y el mantenimiento a largo plazo.

Para este caso concreto usaremos el lenguaje de definición de OpenAPI. Para conectar el API con AWS Lambda a través de API Gateway, se puedes utilizar las extensiones específicas de API Gateway en un archivo de definición de OpenAPI.

Primero, se necesita añadir la URI de la función Lambda en la sección x-*amazon-apigateway-integration* del archivo OpenAPI. Esta sección permite especificar la integración de backend del API. Para la API de inventario, el archivo de definición de OpenAPI sería el siguiente, en formato yaml:

```yaml
openapi: 3.0.0
info:
  title: Inventario API
  version: 1.0.0
paths:
  /productos:
    get:
      summary: Lista todos los productos
      operationId: listaProductos
      responses:
        '200':
          description: Una lista de productos
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Producto'
        '500':
          description: Error del servidor
      x-amazon-apigateway-integration:
```

```
        uri:
          Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${ListaProductosFunction.Arn}/invocations
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        type: "aws_proxy"
  /productos/{id}:
    get:
      summary: Detalles de un producto
      operationId: obtenerProducto
      parameters:
      - name: id
        in: path
        required: true
        schema:
          type: string
      responses:
        '200':
          description: Detalles de un producto
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Producto'
        '404':
          description: Producto no encontrado
        '500':
          description: Error del servidor
      x-amazon-apigateway-integration:
        uri:
          Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${ObtenerProductoFunction.Arn}/invocations
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        type: "aws_proxy"
components:
  schemas:
    Producto:
      type: object
      properties:
        id:
          type: string
        nombre:
          type: string
        precio:
          type: number
        unidadesDisponibles:
          type: integer
```

Este archivo define dos rutas: una para obtener una lista de todos los productos (GET /productos) y otra para obtener detalles de un producto específico (GET /productos/{id}). Cada ruta tiene respuestas definidas y utiliza el esquema de Producto para definir cómo se ve un producto.

Por otro lado, es importante establecer prácticas de seguridad adecuadas. API Gateway ofrece varias características de seguridad, como la autorización basada en roles con AWS IAM, la rotación de claves con AWS KMS y la protección contra ataques DDoS con AWS WAF.

La implementación de estas características de seguridad en AWS API Gateway puede realizarse de la siguiente manera:

- Autorización basada en roles con AWS IAM: Esto se hace al crear políticas IAM que permiten o deniegan el acceso a los recursos de AWS y luego adjuntando esas políticas a roles IAM. A continuación, se pueden asignar estos roles a los usuarios o servicios que necesitan acceder al API.
  En la consola de AWS IAM, se puede crear una política como la siguiente y adjuntarla a un rol. Esta política otorgaría permiso para invocar la API especificada a cualquier entidad que tenga este rol IAM.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke"
            ],
            "Resource": [
                "arn:aws:execute-api:region:account-id:api-id/*"
            ]
        }
    ]
}
```

- Rotación de claves con AWS KMS: AWS Key Management Service (KMS) es un servicio que te permite crear y administrar claves criptográficas para cifrar y descifrar datos. Se pueden usar estas claves para cifrar los datos en tránsito entre la API y los clientes. La rotación de claves se puede habilitar en la configuración de la clave en KMS. Cuando la rotación de claves está habilitada, AWS KMS genera automáticamente una nueva clave criptográfica cada año.

- Protección contra ataques DDoS con AWS WAF: Se puede usar AWS WAF (Web Application Firewall) para proteger el API Gateway de ataques DDoS. AWS WAF permite definir reglas de seguridad personalizables que bloquean tipos comunes de ataques, como inyección de SQL o scripting entre sitios (XSS). Para usar AWS WAF con API Gateway, simplemente se debe asociar una Web ACL (Access Control List) de AWS WAF al API en la configuración de API Gateway.

Además de esto, también se pueden configurar reglas de limitación de velocidad y cuotas para el API para prevenir el uso abusivo y gestionar mejor tus recursos. API Gateway también ofrece capacidades de monitoreo y registro en profundidad a través de integración con AWS CloudWatch y X-Ray, lo que te permite rastrear y optimizar el rendimiento del API.

### d. Pruebas

En el proceso de transformación de una aplicación monolítica a una arquitectura basada en microservicios, es imperativo realizar pruebas exhaustivas para asegurar que los módulos funcionen correctamente tanto de manera independiente como en conjunto. La realización de pruebas a nivel de unidad, de integración y de carga son esenciales para verificar la funcionalidad, la interoperabilidad y la escalabilidad de cada microservicio.

**Pruebas a nivel de unidad**: Las pruebas unitarias son fundamentales para garantizar que cada componente individual de la aplicación funciona según se espera. En el caso de las funciones Lambda de AWS, las pruebas unitarias pueden ayudar a asegurar que la lógica de la función es correcta y que maneja adecuadamente los diferentes tipos de entrada.

Para probar la función Lambda que maneja las operaciones de inventario, se pueden escribir pruebas unitarias que comprueben cada una de las posibles operaciones: obtener un producto, actualizar el precio y cambiar el número de unidades disponibles.

Para ello, se puede hacer uso de un marco de pruebas como Jest, una popular herramienta de pruebas de JavaScript. Aquí hay un ejemplo de cómo se podrían estructurar estas pruebas:

```
const lambda = require('./lambda_function');
const AWS = require('aws-sdk-mock');
const AWS_SDK = require('aws-sdk');

describe('Inventory Lambda Handler', () => {
  beforeAll(() => {
    process.env.AWS_REGION = 'us-west-2';
  });

  beforeEach(() => {
       AWS.setSDKInstance(AWS_SDK);
  });

  afterEach(() => {
       AWS.restore('DynamoDB.DocumentClient');
  });

  test('getProduct retrieves the correct product', async () => {
    const returnData = { Item: { productId: '1', price: 100, stock: 20 } };
    AWS.mock('DynamoDB.DocumentClient', 'get', (params, callback) => {
        callback(null, returnData);
    });

    const event = {
        httpMethod: 'GET',
        pathParameters: { productId: '1' },
```

```
    };

    const response = await lambda.handler(event);
        expect(response.statusCode).toBe(200);
        expect(JSON.parse(response.body)).toEqual(returnData.Item);
  });

});
```

Este código establece un mock para la respuesta de DynamoDB y verifica que la función Lambda devuelve el producto correcto cuando se realiza una petición GET.

Las pruebas adicionales para updatePrice y updateStock seguirían un patrón similar, pero verificarían que la función Lambda realiza correctamente las operaciones de actualización en la base de datos y devuelve una respuesta adecuada.

Tener pruebas unitarias para cada operación individual en la función Lambda ayuda a asegurar que cada componente funcione correctamente, facilitando así la detección de errores antes de que la función se despliegue en el entorno de producción.

**Pruebas de integración:** Las pruebas de integración son esenciales para garantizar que todos los componentes de la arquitectura serverless funcionan correctamente juntos. A diferencia de las pruebas unitarias, las pruebas de integración se centran en la funcionalidad del sistema como un todo, y no solo en sus componentes individuales.

En el contexto de nuestra aplicación, las pruebas de integración pueden involucrar verificar que las peticiones a la API Gateway resultan en el comportamiento esperado de las funciones Lambda y que los datos se manejan correctamente a lo largo del proceso. Por ejemplo, podríamos querer

verificar que una petición para actualizar el precio de un producto resulta efectivamente en un cambio en la base de datos.

Una herramienta útil para realizar pruebas de integración en AWS es el framework Serverless Artillery. Este framework permite definir escenarios de prueba y generar cargas de trabajo para probar la funcionalidad y rendimiento de las aplicaciones serverless.

Aquí hay un ejemplo de cómo se podrían definir las pruebas de integración para nuestro módulo de inventario:

```
config:
  target: "https://api-gateway-url/"
  phases:
   - duration: 60
   arrivalRate: 5

scenarios:
  - flow:
   - get:
       url: "/inventory/{productId}"
       expect:
         - statusCode: 200
         - hasProperty: ["productId", "price", "stock"]
  - put:
       url: "/inventory/{productId}"
       json:
         price: 200
       expect:
         - statusCode: 200
  - get:
       url: "/inventory/{productId}"
       expect:
         - statusCode: 200
         - hasProperty: ["productId", "price", "stock"]
         - equals:
                - jsonpath: "$.price"
               value: 200
```

Este escenario de prueba realizará una petición GET para obtener un producto, luego actualizará el precio del producto con una petición PUT, y finalmente comprobará que el precio se ha actualizado correctamente con otra petición GET. Serverless Artillery proporciona una forma eficiente y efectiva de realizar pruebas de integración en aplicaciones serverless.

Las pruebas de integración son una parte esencial del proceso de desarrollo de aplicaciones serverless. Proporcionan una mayor confianza en la funcionalidad de la aplicación al garantizar que todos los componentes del sistema funcionan correctamente en conjunto.

**Pruebas de carga:** Las pruebas de carga forman parte esencial del ciclo de vida del desarrollo de una aplicación, especialmente en el contexto de una arquitectura serverless. Estas pruebas permiten identificar cómo se comportará la aplicación bajo condiciones extremas de uso, es decir, cuando hay un número elevado de solicitudes simultáneas.

En el marco de la arquitectura serverless, este tipo de pruebas es aún más relevante, ya que uno de los principales beneficios de la tecnología serverless es su capacidad para escalar automáticamente de acuerdo con la demanda. Por lo tanto, es fundamental verificar que este escalado se produce de forma correcta y eficiente.

Podemos seguir utilizando Serverless Artillery para llevar a cabo estas pruebas de carga. El propósito de estas pruebas será determinar cómo se comporta nuestra aplicación serverless bajo un volumen alto de solicitudes.

Aquí hay un ejemplo de cómo se podría configurar Serverless Artillery para realizar pruebas de carga en nuestro módulo de inventario:

```
config:
  target: "https://api-gateway-url/"
  phases:
   - duration: 300
   arrivalRate: 20
```

```
  rampTo: 100
scenarios:
  - flow:
  - get:
      url: "/inventory/{productId}"
  - put:
      url: "/inventory/{productId}"
      json:
           price: 200
```

En este caso, el escenario de prueba consiste en realizar una serie de peticiones GET y PUT a la API del módulo de inventario. La configuración de las fases establece que la prueba de carga durará 300 segundos (5 minutos), con una tasa de llegada inicial de 20 solicitudes por segundo que se incrementará gradualmente hasta llegar a 100 solicitudes por segundo.

Con estas pruebas, podremos observar cómo se comporta la aplicación a medida que aumenta el número de solicitudes, si las funciones Lambda se escalan correctamente para manejar el aumento de la carga, y si la base de datos es capaz de manejar el incremento de lecturas y escrituras.

Estos resultados nos proporcionarán información valiosa sobre la capacidad de nuestra aplicación para manejar altas cargas de trabajo y nos permitirán identificar y corregir posibles cuellos de botella antes de que la aplicación se despliegue en un entorno de producción.

Como hemos visto, para facilitar la realización de estas pruebas, existen numerosas herramientas y frameworks disponibles, como Mocha o Jest para pruebas a nivel de unidad en Node.js, Postman, Artillery o RestAssured para pruebas de API y Artillery, Gatling o Apache JMeter para pruebas de carga. Además, servicios como AWS X-Ray pueden proporcionar trazabilidad y facilitar la depuración de las aplicaciones serverless en AWS. La ejecución de estas pruebas debe integrarse en el proceso de desarrollo y despliegue continuo (CI/CD) para detectar y corregir cualquier problema lo más pronto posible.

### e. Despliegue de la Arquitectura Serverless

**Desplegar Módulos:**

Desplegar los módulos en una plataforma serverless como AWS implica varias etapas y herramientas. En este caso, usaremos el Framework Serverless para el despliegue, que nos proporciona un entorno sencillo y efectivo para gestionar y desplegar funciones serverless.

El Serverless Framework es una herramienta de código abierto que permite a los desarrolladores construir y desplegar aplicaciones serverless de manera fácil y eficiente. Fue uno de los primeros marcos que salieron al mercado para manejar la arquitectura serverless y desde entonces ha ganado mucha popularidad debido a su simplicidad y poder.

El Framework Serverless simplifica el proceso de desarrollo de aplicaciones serverless al proporcionar una interfaz de línea de comandos (CLI) intuitiva y un archivo de configuración simple (*serverless.yml*) que define tus funciones, eventos y recursos.

En primer lugar, es necesario instalar Node.js, NPM y el Framework Serverless en el entorno de desarrollo. Para instalar Serverless se utilizará el comando NPM:

```
npm install -g serverless
```

Es necesario tener las credenciales de AWS configuradas correctamente en el entorno de desarrollo. Para hacerlo se usa comando AWS CLI:

```
aws configure
```

El siguiente archivo describe la estructura básica de un archivo *serverless.yml*, que nos permite define la infraestructura de nuestra aplicación serverless.

```
service: inventory-service
```

```yaml
provider:
  name: aws
  runtime: nodejs14.x
  region: us-east-1
  stage: ${opt:stage, 'dev'}

functions:
  getInventory:
    handler: handler.getInventory
    events:
      - http:
          path: inventory/{id}
          method: get
  updateInventory:
    handler: handler.updateInventory
    events:
      - http:
          path: inventory/{id}
          method: put

resources:
  Resources:
    InventoryTable:
      Type: 'AWS::DynamoDB::Table'
      DeletionPolicy: Retain
      Properties:
        TableName: ${self:provider.stage}-InventoryTable
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
```

Este archivo YAML define dos funciones serverless (getInventory y updateInventory), cada una de las cuales se desencadena por un evento HTTP

específico en la API Gateway. También se define un recurso para la base de datos DynamoDB que almacena la información de inventario.

Una vez que se tiene el archivo configurado, se puede desplegar el servicio serverless utilizando el comando serverless deploy:

```
sls deploy --stage dev
```

Este comando desplegará el servicio en AWS y proporcionará la URL de la API Gateway donde el servicio está disponible.

**CI/CD**

La Integración Continua (Continuous Integration, CI) y la Entrega Continua (Continuous Delivery, CD) son prácticas de desarrollo de software que facilitan y automatizan la creación, el despliegue y las pruebas de las aplicaciones.

- Integración Continua (CI): Es la práctica de combinar automáticamente el trabajo de varios desarrolladores en un solo proyecto. Esto implica la automatización de la construcción y las pruebas del software para detectar cualquier problema o error lo más rápido posible. Las herramientas comunes de CI incluyen Jenkins, Travis CI y CircleCI. En el caso de AWS, se puede usar AWS CodeBuild y AWS CodePipeline para la CI.

- Entrega Continua (CD): Es la extensión natural de la Integración Continua; una práctica de desarrollo de software donde se construyen automáticamente nuevas versiones de una aplicación y se preparan para su lanzamiento a producción. En el contexto de las aplicaciones serverless, las herramientas de CD pueden ayudar a automatizar el despliegue de nuevas versiones de las funciones y la infraestructura asociada.

En este punto veremos un ejemplo de cómo configurar un pipeline de CI/CD en AWS:

- Configurar AWS CodeBuild[7]: CodeBuild es un servicio de AWS que compila el código fuente, ejecuta pruebas y produce paquetes de software listos para ser desplegados.

- Configurar AWS CodePipeline[8]: CodePipeline es un servicio de AWS que automatiza las etapas de entrega del software. Se puede configurar un pipeline que escuche los cambios en el código fuente en GitHub o AWS CodeCommit, construya el código y las funciones Lambda con AWS CodeBuild y, finalmente, despliegue el software con AWS CloudFormation o AWS Elastic Beanstalk.

- Automatizar las pruebas: Las pruebas unitarias y de integración se pueden automatizar en la etapa de construcción del pipeline con AWS CodeBuild. Para las pruebas de carga, usaremos los elementos que hemos visto en el punto anterior.

- Desplegar automáticamente: Una vez que se han pasado todas las pruebas, el software se puede desplegar automáticamente. AWS CodePipeline puede desplegar las funciones Lambda y las API Gateway utilizando AWS CloudFormation.

Este proceso asegura que cualquier cambio en el código fuente sea probado y desplegado automáticamente, minimizando los errores y acelerando el ciclo de entrega del software.

El siguiente diagrama muestra un flujo típico de integración y entrega continua en AWS, utilizando servicios como AWS CodeCommit, AWS CodeBuild, AWS CodeDeploy y AWS CodePipeline:

[7] https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html
[8] https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html

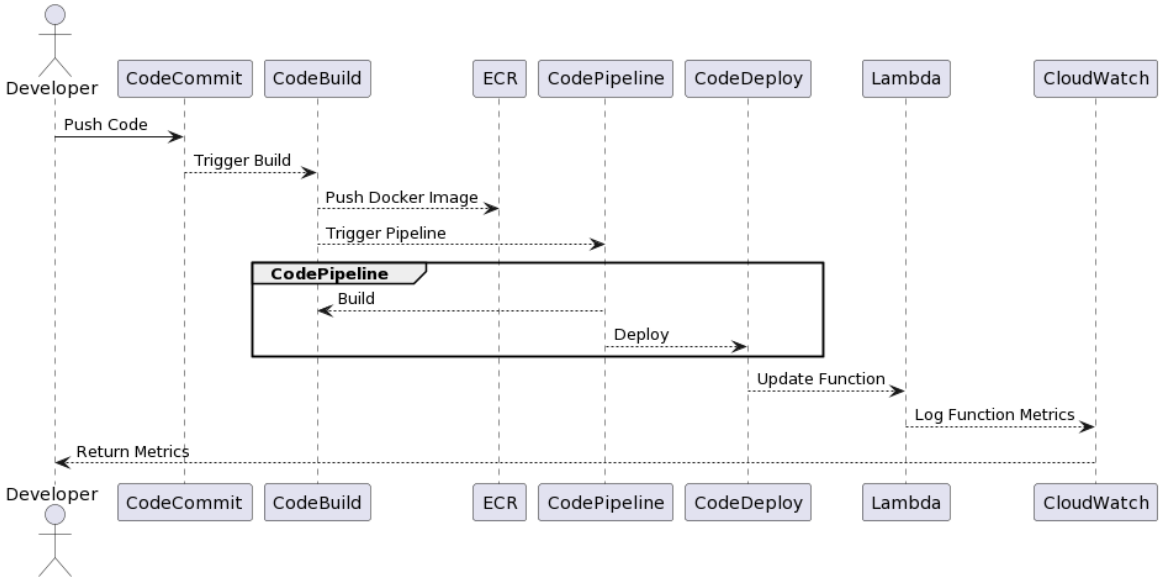



*Ilustración 11  Diagrama CI/CD*

Este diagrama comienza con un desarrollador que sube el código a AWS CodeCommit, el servicio de control de versiones de código fuente. Este push de código desencadena un proceso de compilación en AWS CodeBuild, que crea una imagen Docker del código y la empuja a AWS Elastic Container Registry (ECR).

La compilación también desencadena un pipeline en AWS CodePipeline, que gestiona la entrega continua del código. El pipeline incluye fases de compilación y despliegue, que son gestionadas por CodeBuild y CodeDeploy respectivamente.

El código se despliega a AWS Lambda como una nueva versión de una función. Una vez que la función se ha actualizado, AWS CloudWatch registra métricas de la función, que se devuelven al desarrollador.

### f. Monitoreo y Ajustes.

El monitoreo y los ajustes constituyen una parte integral del despliegue de una arquitectura serverless. Posterior al lanzamiento de los módulos en la plataforma serverless seleccionada, es crucial monitorear el rendimiento de la nueva arquitectura. Este proceso implica recopilar y analizar métricas de rendimiento y uso para identificar áreas de mejora y posibles problemas.

AWS proporciona varias herramientas para monitorear y ajustar aplicaciones serverless, incluyendo AWS CloudWatch, AWS X-Ray y AWS Lambda Insights.

AWS CloudWatch permite recopilar y visualizar métricas de rendimiento, configurar alarmas basadas en métricas y realizar un seguimiento de los logs de las aplicaciones. Este servicio puede ser utilizado para monitorizar varias métricas de AWS Lambda, como la duración de las invocaciones, los errores y las invocaciones concurrentes.

Por su parte, AWS X-Ray proporciona trazas de peticiones a través de la aplicación, lo que permite identificar cuellos de botella y problemas de rendimiento. X-Ray permite visualizar y analizar cómo fluyen las solicitudes a través de la aplicación, lo que puede ayudar a optimizar las interacciones entre los microservicios.

Para habilitar el rastreo con AWS X-Ray para API Gateway, se debe ir a la configuración de la API en la consola de API Gateway y habilitar el rastreo de X-Ray.

Una vez que el rastreo está habilitado, se pueden ver las trazas de las solicitudes en la consola de X-Ray. Cada traza muestra la ruta que tomó una solicitud a través de tu aplicación. Esto incluirá detalles sobre cuánto tiempo pasó la solicitud en cada servicio, y te permitirá identificar fácilmente dónde se pueden estar produciendo cuellos de botella o retrasos.

Éste es un ejemplo de una traza de AWS X-Ray para una solicitud que pasa a través de una API Gateway y luego a una función Lambda:

```
{
  "TraceId": "1-581cf771-a006649127e371903a2de979",
  "Segments": [
  {
      "Id": "a1b2c3d4",
      "Document": {
    "name": "APIGateway",
       "start_time": 1587329964.478,
    "end_time": 1587329964.588,
    "http": {
      "request": {
           "method": "GET",
           "client_ip": "192.0.2.0",
         "url": "https://api.example.com/items/72",
           "user_agent": "Mozilla/5.0",
           "x_forwarded_for": true,
           "traced": true
      },
         "response": {
           "status": 200,
           "content_length": 294
      }
    }
  }
  },
  {
  "Id": "e5f6g7h8",
  "Document": {
    "name": "AWSLambda",
       "start_time": 1587329964.478,
    "end_time": 1587329964.588,
    "aws": {
       "resource": "myLambdaFunction",
      "type": "AWS::Lambda::Function",
      "request_id": "c2a2515a-5a78-48a0-8c3f-2163c0854170"
    }
  }
  }
  ]
}
```

La visualización de estas trazas en la consola de AWS X-Ray proporciona una visión clara del flujo de solicitudes a través de tu aplicación, permitiéndote identificar y solucionar problemas de rendimiento.

AWS Lambda Insights, una característica de CloudWatch, ofrece un análisis más profundo del rendimiento de las funciones Lambda. Permite monitorear y visualizar detalles de rendimiento de las funciones Lambda, como la utilización de la CPU, la memoria, el tiempo de ejecución y otros parámetros relevantes.

Una vez que se han recogido y analizado estas métricas, pueden realizarse ajustes para optimizar el rendimiento y la eficiencia de la arquitectura serverless. Estos ajustes pueden incluir la optimización del código, la configuración de la función Lambda (como el tiempo de ejecución y la memoria asignada) y la configuración de la API Gateway.

### g. Optimización y Escalado

**Optimización**: La optimización es una fase crucial en el ciclo de vida de cualquier arquitectura serverless. Esta fase se centra en mejorar la eficiencia y rendimiento de los servicios serverless para ofrecer una experiencia de usuario mejorada y reducir los costos. La optimización puede abordar varias áreas, como la reducción de la latencia, el aumento del rendimiento, la mejora de la escalabilidad y la eficiencia de costos.

- Optimización de la latencia y el rendimiento: Una forma de mejorar la latencia y el rendimiento es utilizar el modelado de la arquitectura. Por ejemplo, se pueden usar arquitecturas de múltiples regiones en AWS para colocar servicios más cerca de los usuarios finales y reducir la latencia. Además, se pueden optimizar las funciones AWS Lambda ajustando el tiempo de ejecución y la asignación de memoria para asegurar que las funciones se ejecutan de manera eficiente.

- Escalabilidad: La optimización de la escalabilidad puede implicar el ajuste de la configuración de auto-escalado de los servicios. En AWS Lambda, por ejemplo, se puede ajustar la configuración de concurrencia para permitir que las funciones se escalen correctamente bajo cargas pesadas.

- Eficiencia de costos: La optimización de costos puede implicar el uso de herramientas de monitorización y análisis para identificar y eliminar los recursos subutilizados. También puede incluir la revisión y optimización de las políticas de retención de logs y datos.

- APIs: Las APIs deben ser diseñadas para ser altamente disponibles y escalables. La implementación de APIs con Amazon API Gateway permite la creación de APIs RESTful y WebSocket que pueden servir a millones de solicitudes por segundo. Adicionalmente, API Gateway ofrece capacidades de caché para mejorar la latencia y reducir la cantidad de solicitudes dirigidas a los puntos finales de las APIs, lo que puede resultar en una mejora del rendimiento y una reducción de costos.

- Diseño de la aplicación: Es esencial seguir las mejores prácticas de diseño de aplicaciones serverless, como el "Principio de Responsabilidad Única" (una función Lambda debería hacer una cosa y hacerla bien) y evitar los anti-patrones de serverless (por ejemplo, el uso inadecuado de funciones Lambda para tareas largas).

Como vemos, la optimización de una arquitectura serverless implica un enfoque holístico que considera aspectos tanto de infraestructura como de diseño de aplicaciones. A través de la monitorización constante y del análisis del rendimiento del sistema, es posible identificar áreas de mejora y aplicar cambios incrementales para mejorar la eficiencia, el rendimiento y la escalabilidad del sistema.

**Escalado**: El escalado es una capacidad inherente a la arquitectura serverless y es uno de sus aspectos más atractivos, ya que permite que una aplicación pueda manejar eficientemente fluctuaciones en la demanda sin

requerir intervención manual. En este contexto, la arquitectura serverless de AWS proporciona diferentes técnicas para el escalado de aplicaciones, que se adaptan a las necesidades cambiantes del negocio.

En el caso de las funciones AWS Lambda, el escalado es automático. Cada vez que se recibe una solicitud, Lambda se encarga de ejecutar la función en un entorno aislado, de modo que cada función se ejecuta en paralelo para atender todas las solicitudes entrantes. Esto significa que, si una función Lambda está configurada para manejar el módulo de inventario, por ejemplo, y se reciben varias solicitudes para acceder o modificar el inventario, Lambda manejará automáticamente el escalado para atender todas las solicitudes simultáneamente.

En el caso de las APIs desarrolladas con Amazon API Gateway, este servicio administra todo el ciclo de vida de las APIs, desde su creación hasta su monitoreo y escalado. API Gateway puede manejar miles de solicitudes concurrentes y también puede escalar automáticamente para responder a cambios en el tráfico.

Además de este escalado automático, AWS ofrece la opción de configurar el Auto Scaling para ajustar de manera dinámica la capacidad de las bases de datos o los grupos de seguridad en función de las condiciones definidas por el usuario.

Sin embargo, es importante notar que, aunque el escalado es automático en una arquitectura serverless, también puede ser necesario optimizar el rendimiento de las funciones y las APIs para garantizar que el escalado sea eficiente. Por ejemplo, puede ser útil minimizar el tiempo de inicio en frío de las funciones Lambda, o reducir la latencia de las APIs utilizando una integración de tipo proxy.

En términos prácticos, esto implica seguir buenas prácticas de desarrollo, como mantener el código de las funciones lo más ligero posible, utilizar correctamente las conexiones a la base de datos y gestionar las dependencias de manera eficiente. Como hemos visto en el punto anterior, también puede ser beneficioso monitorear de cerca el rendimiento de las funciones y las APIs

utilizando herramientas como AWS X-Ray y CloudWatch, para identificar y solucionar cualquier problema de rendimiento.

Por último, el escalado eficiente de una arquitectura serverless también implica consideraciones de costos. Aunque el escalado automático puede ayudar a manejar picos de demanda, también puede resultar en costos más altos. Por lo tanto, es importante configurar alertas para el monitoreo del uso y los costos, y considerar la posibilidad de utilizar reservas de capacidad para las funciones Lambda si se espera un tráfico constante.

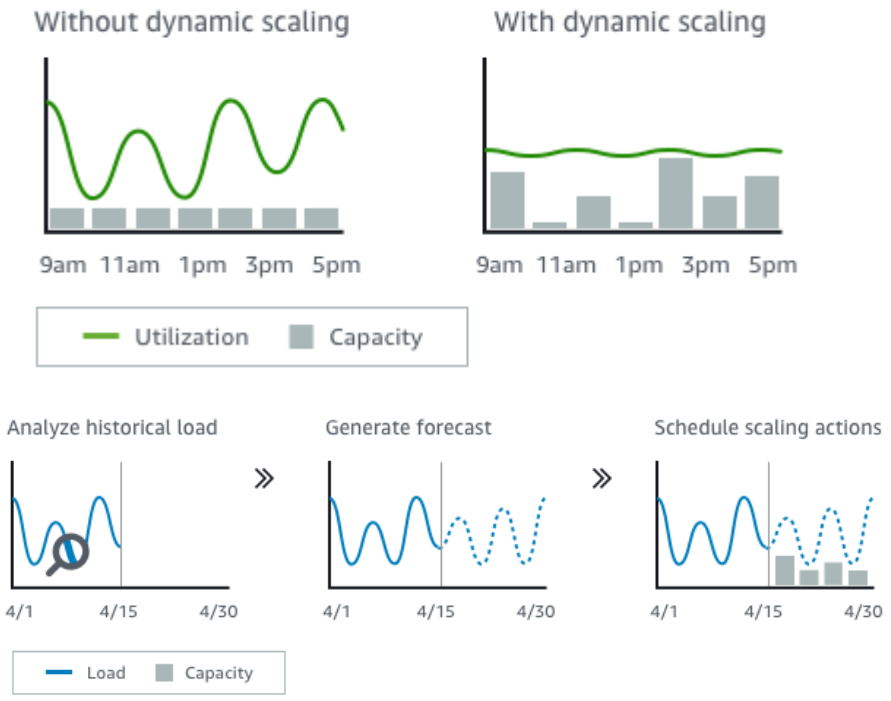



Ilustración 12 Ejemplos de escalados (Dinámico y Predictivo)[9]

[9] https://docs.aws.amazon.com/autoscaling/plans/userguide/how-it-works.html

# 5. Evaluación y resultados

## a. Evaluación de la eficiencia y flexibilidad

La implementación de la arquitectura serverless utilizando AWS Lambda y Amazon API Gateway ha demostrado ser altamente eficiente y flexible. Gracias al diseño modular de la arquitectura, cada microservicio puede desarrollarse, probarse y desplegarse de manera independiente, lo que agiliza el proceso de desarrollo y facilita la resolución de problemas y la introducción de cambios. Además, el escalado automático de AWS Lambda y API Gateway permite manejar fluctuaciones en la demanda sin requerir intervención manual, lo que mejora la eficiencia y la disponibilidad del sistema.

Las siguientes métricas nos pueden dar información acerca de la mejora en la eficiencia de la aplicación, una vez modularizada:

- Tiempo de respuesta: Medido en milisegundos, el tiempo que toma para un servicio responder a una solicitud. Un valor más bajo indica una mayor eficiencia.

- Rendimiento: Medido en solicitudes por segundo, cuántas solicitudes puede manejar un servicio. Un valor más alto indica una mayor eficiencia.

- Tiempo de escalado: Medido en segundos, cuánto tiempo toma para que el servicio escale en respuesta a un aumento en la demanda. Un valor más bajo indica una mayor flexibilidad.

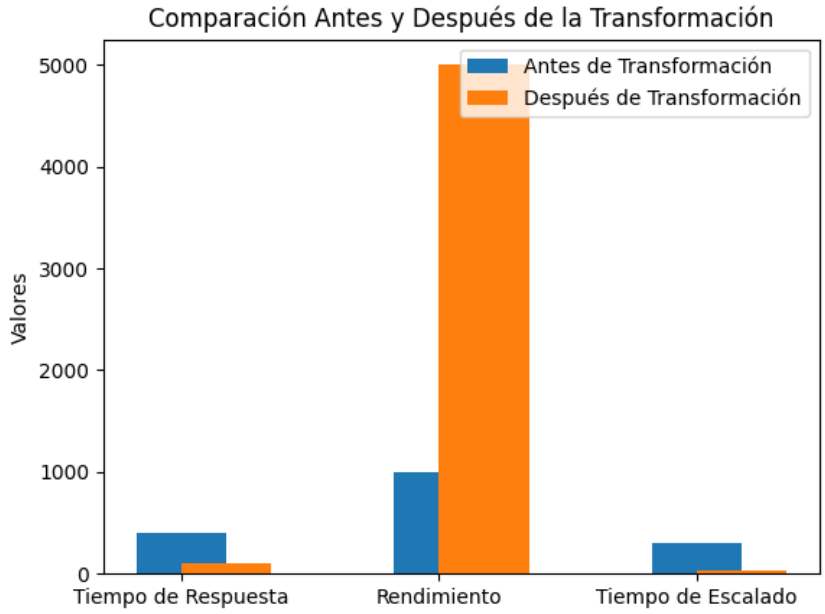



Ilustración 13 Métricas de eficiencia y rendimiento después de la transformación

### b. Comparativa antes y después de la transformación tecnológica

Antes de la transformación, la aplicación monolítica presentaba problemas de escalabilidad y mantenimiento. La modificación de una funcionalidad o la introducción de una nueva característica requería una comprensión completa del código base, lo que incrementaba la complejidad y el tiempo de desarrollo. Además, el rendimiento de la aplicación dependía de un único servidor, lo que limitaba su capacidad para manejar picos de demanda.

Después de la transformación, cada módulo se desarrolla y despliega de forma independiente, lo que mejora la eficiencia del desarrollo y permite un mayor grado de flexibilidad. El uso de AWS Lambda y API Gateway permite el escalado automático de la aplicación para atender eficientemente las

fluctuaciones de la demanda, lo que mejora la disponibilidad y la experiencia del usuario.

Podemos ver esta comparativa de forma más clara en la siguiente tabla:

| Aspectos | Antes de la transformación | Después de la transformación |
|---|---|---|
| **Escalabilidad** | La escalabilidad era limitada debido a la naturaleza monolítica de la aplicación. La aplicación dependía de un único servidor, lo que limitaba su capacidad para manejar picos de demanda. | La escalabilidad se mejoró significativamente gracias al escalado automático de AWS Lambda y API Gateway. La aplicación ahora puede manejar eficientemente las fluctuaciones de la demanda. |
| **Mantenimiento** | La modificación de una funcionalidad o la introducción de una nueva característica requería una comprensión completa del código base, lo que incrementaba la complejidad y el tiempo de desarrollo. | Cada microservicio se puede desarrollar, probar y desplegar de forma independiente, lo que mejora la eficiencia del desarrollo y reduce el tiempo necesario para introducir cambios. |
| **Costo Operacional** | El costo operacional era alto debido al mantenimiento del servidor y a la necesidad de escalar manualmente la aplicación para atender la demanda. | El uso de la arquitectura serverless reduce los costos operacionales, ya que sólo se paga por el tiempo de ejecución de las funciones y el escalado es automático. |
| **Flexibilidad** | La aplicación monolítica era menos flexible, ya que cualquier cambio requería una comprensión completa del código base y podía afectar a otras partes de la aplicación. | La arquitectura serverless ofrece un alto grado de flexibilidad, ya que cada microservicio se puede desarrollar, probar y desplegar de forma independiente. Esto permite introducir cambios con menor riesgo de afectar a otras partes de la |

| | | |
|---|---|---|
| | | aplicación. |
| **Disponibilidad** | La disponibilidad dependía de la capacidad del servidor. Durante los picos de demanda, la aplicación podía experimentar tiempos de respuesta más lentos o incluso caídas. | La disponibilidad se mejoró gracias al escalado automático de AWS Lambda y API Gateway, que pueden ajustarse rápidamente a las fluctuaciones de la demanda. Esto mejora la experiencia del usuario y asegura un funcionamiento continuo de la aplicación. |

### c. Beneficios y desafíos en el proceso de transformación

Entre los principales beneficios de la transformación está la mejora en la eficiencia del desarrollo, el escalado automático de la aplicación y la reducción de los costos operativos. Además, la arquitectura serverless ofrece un alto grado de flexibilidad, ya que cada microservicio se puede desarrollar, probar y desplegar de forma independiente. Todos estos puntos se han venido analizando a lo largo de todo el trabajo.

En cuanto a los desafíos, el proceso de descomposición de la aplicación monolítica en microservicios requiere una comprensión profunda de la aplicación existente y un diseño cuidadoso de la nueva arquitectura. Asimismo, la implementación de pruebas adecuadas para microservicios y la gestión de la seguridad en una arquitectura serverless pueden presentar desafíos adicionales. Sin embargo, estos desafíos pueden mitigarse mediante la adopción de buenas prácticas de desarrollo y la utilización de las herramientas y servicios que ofrece AWS para la gestión de microservicios y la seguridad en la nube.

Los desafíos genéricos a tener en cuenta en las aplicaciones basadas en APIs y microservicios, lo revisaremos en el punto siguiente.

# 6. Conclusiones y futuras investigaciones

## a. Conclusiones

A lo largo de este trabajo, se ha abordado la problemática de las aplicaciones monolíticas, su impacto en la eficiencia operativa de las empresas y las limitaciones que presentan en términos de escalabilidad, mantenimiento y adaptabilidad al cambio. A su vez, se ha propuesto una solución a estos retos mediante la transformación hacia un ecosistema de aplicaciones modulares bajo un paradigma serverless y de gestión de APIs.

Las arquitecturas serverless, como se ha expuesto, ofrecen una serie de ventajas considerables en comparación con las infraestructuras tradicionales basadas en servidores. Estos beneficios, entre los que se incluyen la escalabilidad automática, la alta disponibilidad, el pago por uso y la eliminación de la necesidad de gestionar servidores, pueden permitir a las empresas mejorar su eficiencia operativa, agilizar el tiempo de lanzamiento al mercado y responder con mayor efectividad a las necesidades cambiantes del negocio.

En el contexto del API Management, se han explorado sus fundamentos, beneficios y alternativas en el mercado. A través de su implementación, se ha demostrado cómo puede facilitar la integración de aplicaciones, mejorar la seguridad y controlar el acceso, proporcionando una interfaz estandarizada y simplificada para interactuar con los servicios de backend.

El caso práctico desarrollado ha servido para ilustrar de manera concreta cómo puede llevarse a cabo el proceso de transformación de una aplicación monolítica hacia una arquitectura serverless. En este proceso, se han detallado las etapas de análisis y planificación, desacoplamiento de la aplicación monolítica, pruebas, despliegue, monitoreo y ajustes, así como la optimización y escalado de la arquitectura.

Los resultados obtenidos a lo largo del estudio han evidenciado la eficiencia y flexibilidad de las soluciones implementadas. La comparación de los resultados antes y después de la transformación tecnológica ha puesto de

manifiesto las mejoras en términos de tiempo de respuesta, rendimiento y tiempo de escalado.

No obstante, aunque los beneficios de las arquitecturas serverless y la gestión de APIs son evidentes, también se han identificado desafíos en el proceso de transformación. Entre ellos se incluyen la necesidad de reestructurar el código existente, la complejidad de las pruebas en un entorno serverless y los posibles problemas de latencia en las funciones frías.

Finalmente, este trabajo sienta las bases para futuras investigaciones en diversas direcciones. Por un lado, se podría profundizar en la comparación de diferentes proveedores de servicios serverless y API Management, analizando en detalle sus características, costos y rendimiento. Por otro lado, se podrían explorar técnicas avanzadas de optimización y escalado para mejorar aún más la eficiencia de las arquitecturas serverless. Asimismo, se podría investigar la aplicabilidad de estas soluciones en diferentes dominios y sectores, evaluando su impacto en términos de productividad, rentabilidad y satisfacción del cliente.

### b. Tendencias y desafíos futuros

La gestión de API ha experimentado cambios significativos en los últimos años debido al rápido avance de la tecnología y la creciente necesidad de digitalización en empresas de todos los sectores. Aquí se analizan algunas de las tendencias emergentes más importantes en la gestión de API, que podrían ser tenidas en consideración para futuras investigaciones:

- **API como producto:** Las APIs ya no se ven solo como interfaces para la integración de sistemas, sino como productos en sí mismos que pueden generar valor de negocio. Las empresas están empezando a asignar equipos de producto dedicados a sus APIs, con roles como gestores de producto de API, diseñadores de API y defensores del desarrollador.

- **API-first Design:** El diseño API-first, o "API primero", es una filosofía que prioriza la creación de la API antes que cualquier otro componente de un sistema. Este enfoque ayuda a garantizar que las

APIs son coherentes, escalables y fácilmente consumibles por otros desarrolladores.

- **API para IA y aprendizaje automático:** El uso de APIs para facilitar el acceso a capacidades de inteligencia artificial (IA) y aprendizaje automático (ML) está creciendo. Estas APIs permiten a las empresas aprovechar algoritmos complejos sin la necesidad de desarrollar la experiencia interna.

- **GraphQL:** GraphQL está ganando popularidad como alternativa a REST para la creación de APIs. Permite a los clientes especificar exactamente qué datos necesitan, lo que puede mejorar el rendimiento y hacer que la API sea más fácil de usar para los desarrolladores.

Por otro lado, la gestión de API enfrenta desafíos significativos en el futuro, pero estos desafíos también presentan oportunidades para la innovación y el crecimiento.

- **Seguridad de API:** Con el creciente número de ataques a APIs, la seguridad se ha convertido en un desafío crítico. Sin embargo, esto también representa una oportunidad para desarrollar soluciones más seguras y robustas para la gestión de API.

- **Interoperabilidad:** A medida que las empresas adoptan cada vez más una variedad de tecnologías de diferentes proveedores, la interoperabilidad de las APIs se ha convertido en un problema. Aquí se presenta una oportunidad para desarrollar estándares y soluciones que faciliten la integración entre diferentes sistemas y APIs.

- **Automatización de la gestión de API:** La automatización es una oportunidad importante para mejorar la eficiencia y la consistencia en la gestión de API. Esto podría incluir la automatización del diseño de la API, la implementación y el monitoreo.

- **Economía de API:** A medida que las APIs se vuelven más críticas para los negocios, veremos un mayor desarrollo de la "economía de

API". Esto presenta oportunidades para la creación de nuevos modelos de negocio basados en el uso y la monetización de APIs.

La gestión de API es un campo en constante evolución con un futuro prometedor. Las tendencias actuales indican un enfoque cada vez mayor en la consideración de las API como productos y en la seguridad de las API. Al mismo tiempo, la creciente necesidad de interoperabilidad y la emergencia de la economía de API representan desafíos y oportunidades significativas para el futuro de la gestión de API.

Este trabajo, además de revisar un caso clásico de modularización de una aplicación monolítica aprovechando las ventajas y capacidades que nos proporciona una correcta gestión e implementación de una capa de API, ha explorado las tendencias y desafíos de éstas, y cómo pueden influir en la evolución de la gestión de API en el futuro.

## 7. Referencias


1. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2016). Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering, pp. 195-216.
2. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. J., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. (2017). Serverless Computing: Economic and Architectural Impact. Communications of the ACM, 60(12), 37-39.
3. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Rabbah, R. (2017). Serverless Computing: Current Trends and Open Problems. En Research Advances in Cloud Computing (pp. 1-20). Springer, Singapore.
4. Lewis, J., & Fowler, M. (2014). Microservices: A definition of this new architectural term. ThoughtWorks.
5. Roberts, M. (2018). Serverless Architectures. MartinFowler.com.
6. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.
7. Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. En European Conference on Service-Oriented and Cloud Computing (pp. 137-151). Springer, Cham.
8. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Stoica, I. (2019). Cloud programming simplified: A Berkeley view on serverless computing. arXiv:1902.03383.
9. Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., & Wu, C. (2018). Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651.
10. Li, Y., Lin, Y., Wang, Y., Ye, K., & Xu, C. (2020). A comprehensive survey on serverless: Research issues, commercial products, and future directions. ACM Computing Surveys (CSUR), 53(6), 1-40. Enlace
11. Malawski, M., Figiela, K., Bubak, M., & Gajek, A. (2019). Serverless execution of scientific workflows: Experiments with hyperflow, AWS Lambda and Google Cloud Functions. Future Generation Computer Systems, 110, 502-514.
12. Sbarski, P. (2017). Serverless Architectures on AWS: With examples using AWS Lambda. Manning Publications.

13. Peper, B., Clinton, D. (2019). AWS Certified Solutions Architect Study Guide: Associate SAA-C01 Exam, Second Edition. Sybex.
14. Haishi, B., Chou, Y., & Read, J. (2019). Architecting Microsoft Azure Solutions - Exam Guide 70-535: A complete guide to passing the 70-535 Architecting Microsoft Azure Solutions exam. Packt Publishing.
15. Collier, M. (2018). Microsoft Azure Essentials: Fundamentals of Azure, Second Edition. Microsoft Press.
16. Lakshmanan, V. (2019). Data Science on the Google Cloud Platform: Implementing End-to-End Real-Time Data Pipelines: From Ingest to Machine Learning. O'Reilly Media.
17. Deng, L. (2020). Amazon Web Services in Action. Manning Publications.
18. Gupta, M., Dastjerdi, A. V., Ghosh, S. K., & Buyya, R. (2017). Fog Computing: Principles, Architectures, and Applications. Morgan Kaufmann.
19. Jassy, A. (2019). AWS: Overview of Amazon Web Services. Amazon Web Services, Inc.
20. Li, A., Yang, X., Kandula, S., & Zhang, M. (2018). CloudCmp: Comparing Public Cloud Providers. ACM SIGCOMM Computer Communication Review, 40(5), 1-14.
21. Sultan, N. (2011). Reaching for the “cloud”: How SMEs can manage. International Journal of Information Management, 31(3), 272-278.
22. Chappell, D. (2018). Introducing Microsoft Azure. Microsoft Press.
23. Fryer, B. (2017). Cloud Computing: A Practical Introduction to the Legal Issues. The Computer & Internet Lawyer, 34(1), 1-8.
24. Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., & Khan, S. U. (2015). The rise of “big data” on cloud computing: Review and open research issues. Information Systems, 47, 98-115.
25. Leitao, J. (2020). Building Serverless Applications with Microsoft Azure. Packt Publishing.
26. Radu, I. (2019). Exploring Microsoft Azure and Its Growing Ecosystem. Journal of Information Systems & Operations Management, 13(1), 99-107.
27. Leitner, P., Wittern, E., Spillner, J., & Hummer, W. (2019). A mixed-method empirical study of Function-as-a-Service software development in industrial practice. Journal of Systems and Software, 149, 340-359.
28. Amundsen, M. (2016). RESTful web APIs: services for a changing world. "O'Reilly Media, Inc.".

29. Rodriguez, A., Baez, M., Daniel, F., Casati, F., & Trabucco, J. (2016). REST APIs: a large-scale analysis of compliance with principles and best practices. In Web Engineering (pp. 21-39). Springer, Cham.
30. S. H. Jeon, S. -J. Cha, C. Ramneek, Y. J. Jeong, J. M. Kim and S. Jung, (2019) Deployment and Evaluation of Azalea multi-kernel for manycore, International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2019, pp. 1178-1180, doi: 10.1109/ICTC46691.2019.8939999.
31. Erl, T. (2009), SOA Design Patterns, Pearson.
32. Erl,T. Carlyle, B., Cesare, Balasubramanian, R.. (2012). SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST (1st. ed.). Prentice Hall Press, USA.
33. Comingdeer, R., y Otros. (2023). Performance Efficiency Pillar: AWS Well-Architected Framework. Amazon Web Services. https://docs.aws.amazon.com/pdfs/wellarchitected/latest/performance-efficiency-pillar/wellarchitected-performance-efficiency-pillar.pdf
34. Mergen, B, y Otros. (2023). Cost Optimization Pillar. AWS Well-Architected Framework. Amazon Web Services. https://docs.aws.amazon.com/pdfs/wellarchitected/latest/cost-optimization-pillar/wellarchitected-cost-optimization-pillar.pdf

## 8. Definiciones

Listado de definiciones útiles para contextualizar y entender el trabajo:

1. API Management: Gestión de APIs - Es el proceso de creación, publicación, mantenimiento y control de acceso a las APIs dentro de una organización o en un entorno empresarial, a fin de garantizar una correcta utilización, monitorización y seguridad de estas interfaces.
2. API: Application Programming Interface - Es un conjunto de protocolos, rutinas y herramientas que permiten la comunicación e interacción entre diferentes componentes de software o aplicaciones.
3. AWS: Amazon Web Services. Es una plataforma de servicios de computación en la nube que ofrece una mezcla de infraestructura como servicio (IaaS), plataforma como servicio (PaaS) y paquetes de software empaquetados como servicio (SaaS).
4. Azure: Es una plataforma de servicios en la nube creada por Microsoft para construir, probar, implementar y administrar aplicaciones y servicios a través de centros de datos administrados por Microsoft.
5. CI/CD: Continuous Integration/Continuous Delivery. Es una metodología de desarrollo de software que requiere que los desarrolladores integren el código en un repositorio compartido varias veces al día. Cada integración puede ser verificada por una compilación automática y pruebas automatizadas.
6. Datadog: Es una plataforma de monitoreo y análisis para desarrolladores, operaciones de TI y otros usuarios de tecnología en la nube.
7. Docker: Es una plataforma de código abierto que permite a los desarrolladores y administradores de sistemas construir, empaquetar y distribuir aplicaciones de manera rápida y fácil.
8. ELK Stack: Elasticsearch, Logstash, Kibana. Es una colección de tres productos de código abierto que se utilizan juntos para ayudar a recoger, analizar y visualizar datos en tiempo real.
9. gRPC: Es un marco de trabajo de alto rendimiento desarrollado por Google que permite la comunicación entre servicios. Utiliza Protocol Buffers como lenguaje de interfaz y puede funcionar sobre HTTP/2.

10. HTTP: Hypertext Transfer Protocol. Es el protocolo de red utilizado en la World Wide Web para la transmisión de datos, especialmente HTML. Permite las comunicaciones entre los clientes y los servidores.
11. IT: Information Technology (en inglés) - Equivalente al término "Tecnologías de la Información" en español. Hace referencia a los recursos y sistemas tecnológicos empleados en la gestión, procesamiento y comunicación de la información en entornos empresariales y organizacionales.
12. Jenkins, CircleCI, GitLab CI/CD: Son herramientas de integración y entrega continuas que permiten a los equipos de desarrollo automatizar partes de su proceso de desarrollo de software.
13. Kubernetes: Es un sistema de código abierto para la automatización del despliegue, el escalado y la gestión de aplicaciones en contenedores.
14. New Relic: Es una plataforma de observabilidad en tiempo real que ayuda a los desarrolladores a detectar, entender y resolver problemas de software rápidamente.
15. OWASP: Open Web Application Security Project. Proporciona orientación y recursos para garantizar la seguridad de las API.
16. SIC: Service Interoperability Consortium. Es un consorcio que desarrolló el estándar Service Component Architecture (SCA) para simplificar el desarrollo de aplicaciones de servicios.
17. Splunk: Es una plataforma de software que busca, analiza y visualiza los datos generados por máquinas de sitios web, aplicaciones, servidores y dispositivos móviles.