¿Son los microservicios la mejor opción? Una evaluación de su eficacia y eficiencia frente a los monolitos

Gonzalez Rodrigo Alejandro, Giménez Sergio Agustín, Molina Pualuk, Numa Roy[1]

[1] Fac. de Ciencias Exactas Naturales y Agrimensura, Universidad Nacional del Nordeste, Corrientes

gonzlrodrigo@gmail.com, agustin029g@gmail.com, numamolina19@gmail.com

Resumen. La arquitectura de microservicios ha surgido como una alternativa para el desarrollo de aplicaciones tradicionales basadas en Monolitos, ofreciendo un enfoque altamente modular y escalable. Sin embargo, su implementación y gestión pueden ser complejas sin las herramientas adecuadas. Como parte de un proyecto colaborativo en el marco de un curso sobre "Arquitectura de Microservicios" en la Licenciatura en Sistemas de Información, exploramos el diseño e implementación de estos. Nuestro objetivo es evaluar su eficacia, y eficiencia como una alternativa a los monolitos.

Palabras claves: Microservicios, Desarrollo de Aplicaciones, Escalabilidad, Gestión de Servicios, Monolito.

Abstract. Microservices architecture has emerged as an alternative to traditional monolithic-based application development, offering a highly modular and scalable approach. However, its implementation and management can be complex without the proper tools. As part of a collaborative project within a course on "Microservices Architecture" in the Bachelor's Degree in Information Systems, we explore the design and implementation of these architectures. Our goal is to evaluate their effectiveness and efficiency as an alternative to monoliths.

Keywords: Microservices, Application Development, Scalability, Service Management, Monolith.

1 Introducción

1.1 Aplicaciones monolíticas

Una aplicación monolítica es una aplicación de *software* de un solo nivel en la que se combinan diferentes módulos en un solo programa. Por ejemplo, si se compila una aplicación de comercio electrónico, se espera que la aplicación tenga una arquitectura modular alineada con los principios de programación orientada a objetos (OOP). En

una aplicación monolítica, los módulos se definen mediante una combinación de construcciones del lenguaje de programación (como los paquetes de Java) y artefactos de compilación (como los archivos JAR de Java).[2]

1.2 **Beneficios de la aplicación monolítica**

La arquitectura monolítica es una solución convencional para compilar aplicaciones. Las siguientes son algunas ventajas de adoptar un diseño monolítico para una aplicación:

- Se puede implementar pruebas de extremo a extremo de una aplicación monolítica mediante herramientas como Selenium.
- Para implementar una aplicación monolítica, simplemente se puede copiar la aplicación empaquetada en un servidor.
- Todos los módulos de una aplicación monolítica comparten memoria, espacio y recursos, de modo que se puede usar una sola solución para abordar problemas cruzados, como el registro, el almacenamiento en caché y la seguridad.
- El enfoque monolítico puede proporcionar ventajas de rendimiento, ya que los módulos pueden llamarse entre sí directamente. Por el contrario, los microservicios suelen requerir una llamada de red para comunicarse entre sí.[3]

1.3 **Desafíos de la aplicación monolítica**

En el contexto de aplicaciones monolíticas, las complejidades se intensifican a medida que crecen en tamaño y complejidad, llegando a un punto donde los desafíos superan a los beneficios. La coordinación extensa de cambios y la dificultad para implementar ajustes en sistemas extensos y complejos son evidentes. La integración continua y la implementación continua (CI/CD) se vuelven desafiantes, requiriendo la reimplementación completa y pruebas manuales extensas. La escalabilidad plantea problemas al asignar recursos a módulos conflictivos, y el riesgo de fallos totales debido a errores en un módulo es inherente. Además, la introducción de nuevas tecnologías implica a menudo la reescritura completa de la aplicación, siendo una tarea costosa en tiempo y recursos financieros.[4]

1.4 **Aplicaciones basadas en microservicios**

Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. Los propietarios de estos servicios son equipos pequeños independientes.[1](**Fig**.1)

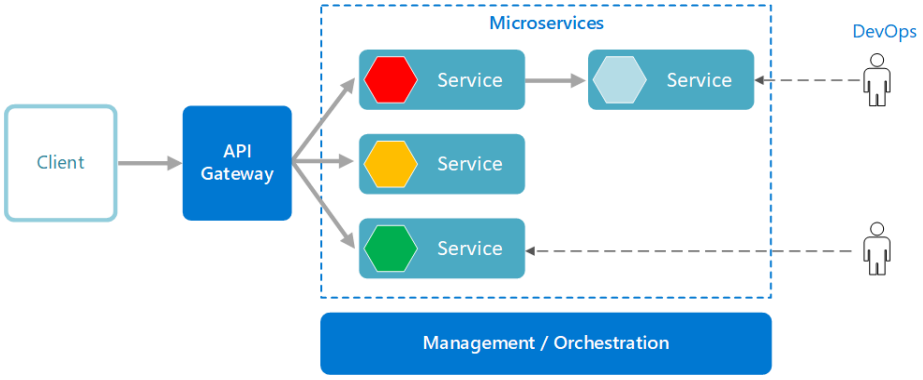

Fig. 1. Se muestra un esquema básico de la arquitectura de microservicios.

1.5 Beneficios de los microservicios

La adopción de la arquitectura de microservicios ofrece ventajas clave al abordar desafíos inherentes a las aplicaciones monolíticas. Al dividir la aplicación en servicios gestionables con límites bien definidos, se facilita el desarrollo y mantenimiento, permitiendo que equipos autónomos trabajen de manera independiente. Esta modularidad posibilita el uso de diferentes lenguajes de programación para cada microservicio, optimizando la aplicación en términos de velocidad, funcionalidad y mantenibilidad. Además, la capacidad de implementar y lanzar servicios de forma independiente mejora la velocidad y la escalabilidad del sistema.

La migración a microservicios resulta beneficiosa para mejorar la escalabilidad, agilidad y velocidad de entrega. También se aplica en la modernización gradual de aplicaciones heredadas y en la extracción de servicios empresariales para su reutilización en distintos canales, contribuyendo a una implementación más eficiente y coherente de funcionalidades comunes.

1.6 Desafíos de los microservicios

La implementación de microservicios, a pesar de sus beneficios, presenta desafíos que requieren una consideración cuidadosa en la arquitectura de aplicaciones. La complejidad de un sistema distribuido, con la necesidad de elegir y aplicar mecanismos de comunicación entre servicios, gestionar fallas y abordar la falta de disponibilidad, destaca como un desafío significativo. La gestión de transacciones distribuidas también surge como una complejidad, ya que las operaciones empresariales distribuidas pueden resultar en la coordinación y reversión complicada de estados en caso de error, potencialmente conduciendo a datos incoherentes.

Las pruebas integrales de aplicaciones basadas en microservicios son más complejas debido a la interacción entre servicios para completar flujos de prueba. La implementación y gestión de una aplicación de microservicios se complica por la multiplicidad de servicios, instancias de entorno y la necesidad de un mecanismo de

descubrimiento de servicios. La sobrecarga operativa aumenta al supervisar y gestionar múltiples servicios, generando alertas y enfrentándose a más puntos de falla en la comunicación servicio a servicio. Aunque la arquitectura permite funciones simultáneas, la latencia se incrementa debido a las llamadas de red entre servicios. Es crucial reconocer que no todas las aplicaciones son adecuadas para la descomposición en microservicios, especialmente aquellas que requieren integración estrecha o procesamiento en tiempo real.[8]

1.7 **Características de los Microservicios**

- Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa. Un único equipo reducido de programadores puede escribir y mantener un servicio.
- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
- Los servicios se comunican entre sí mediante *API* bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- Admite la programación políglota. Por ejemplo, no es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos.[5]

Además de los propios servicios, hay otros componentes que aparecen en una arquitectura típica de microservicios[4]:

Administración e implementación. Este componente es el responsable de la colocación de servicios en los nodos, la identificación de errores, el reequilibrio de servicios entre nodos, etc. Normalmente, este componente es una tecnología estándar, como Kubernetes, en lugar de algo creado de forma personalizada.

Puerta de enlace de *API(API Gateway)*. La puerta de enlace de *API* es el punto de entrada para los clientes. En lugar de llamar a los servicios directamente, los clientes llaman a la puerta de enlace de *API*, que reenvía la llamada a los servicios apropiados en el *back-end*.

2 Instrumentación

Para el despliegue del estudio en cuestión se dispuso de los siguientes paquetes de software y hardware:

- Pc de escritorio con microprocesador Ryzen 5 2600 6 núcleos 12 hilos a 3.4 GHz, 16 GB de ram DDR4 a 2400 Mhz, Caché L1 total 576 KB Caché L2 total 3MB Caché L3 total 16MB
- Fedora 38 Workstation
- Docker
- PostgreSQL
- Apache Jmeter
- Golang 1.20 o superior
- Partición con formato Btrfs
- Grafana
- Prometheus

3 Desarrollo

3.1 Diseño

Para llevar a cabo nuestra investigación, nos centraremos en un monolito que se compone de una *API REST* construida en Golang y usa como base de datos PostgreSQL, diseñada para la gestión de tareas o pendientes. A pesar de que este proyecto es relativamente sencillo, es lo suficientemente complejo como para demostrar la mayoría de los puntos que distinguen un monolito de un microservicio. Primero debemos entender que funciones cumplen que son de manera básica un *CRUD* de usuarios que se puede acceder desde el *endpoint /users* y uno de tareas accesibles desde el *endpoint /taks* y cada módulo con su tabla respectiva en la base datos.(**Fig**.2)

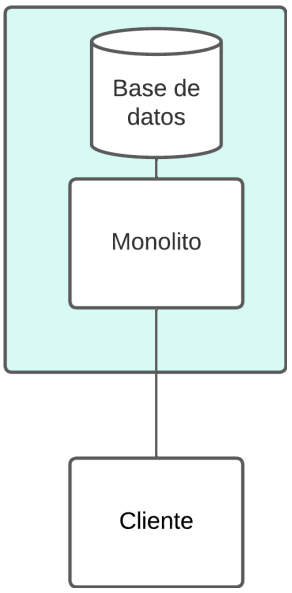

Fig. 2. Grafico del Monolito base.

Dado que los microservicios representan una arquitectura de *software*, es fundamental descomponer nuestro monolito en sus componentes más pequeños y relativamente independientes.
Para eso vamos a seguir los siguientes pasos[9]:

1. Es necesario identificar las partes del código (dominios) que están altamente cohesionadas y bajo acopladas con el resto, puesto que son susceptibles de ser separadas a un microservicio aparte.
2. Separar los dominios en distintos proyectos. Se pueden utilizar lenguajes de programación distintos para cada proyecto, dependiendo de las necesidades de cada dominio. Nuestro consejo es que se trate de homogeneizar, a no ser que haya casos en los que de verdad merezca la pena implementar un dominio en un lenguaje de programación específico.
3. Obtener un grafo de las dependencias existentes en el sistema legado monolítico entre los dominios identificados. Se puede utilizar alguna de las siguientes herramienta que analicen el proyecto:
    a. Dep: Es una herramienta de gestión de dependencias para Go que puede ser útil para analizar las dependencias de un proyecto y mostrar la estructura de las mismas.
    b. GoCallGraph: Esta herramienta genera un gráfico de llamadas para un programa Go, lo que permite visualizar las dependencias entre las diferentes funciones y paquetes en el código.
    c. Go Dep Graph: Es una herramienta de línea de comandos que genera un gráfico de dependencias para un proyecto Go. Se puede usar para visualizar las dependencias entre paquetes y comprender la estructura del código.
    d. JDepend: Esta es una herramienta de análisis de dependencias para proyectos Java. Proporciona un informe detallado de las dependencias entre los paquetes Java en un proyect
    e. Dependency-Check: Es una herramienta de análisis de dependencias de código abierto que detecta y reporta las dependencias de componentes de terceros en aplicaciones Java, .NET, JavaScript y otros lenguajes.
4. A partir de las dependencias identificadas, se debe diseñar una interfaz (siguiendo el enfoque *API first*) para cada uno de los dominios, en este caso, tareas y usuarios. Estas *APIs* pueden ser implementadas utilizando tecnologías como *REST*, GraphQL, gRPC, entre otras, y servirán como la puerta de entrada a cada uno de los dominios. Por ejemplo, si el dominio de tareas necesita acceder a información relacionada con usuarios, deberá utilizar las operaciones proporcionadas por la *API* del dominio de usuarios para obtener los datos necesarios. De esta manera, se establece una clara separación de responsabilidades y una comunicación bien definida entre los diferentes microservicios, favoreciendo la independencia y la cohesión de cada componente funcional.
5. El momento de migrar los datos a los nuevos esquemas de datos de cada uno de los diferentes microservicios. Este paso puede llegar a ser muy costoso, a la par que complicado.

6. Una vez que cada microservicio ha sido simplificado y cumple con su funcionalidad específica, ya sea con la ayuda de una o varias bases de datos, es el momento de integrarlos. Esto implica establecer interacciones entre ellos, reemplazando las dependencias de módulos presentes en el monolito por llamadas a sus respectivas *APIs*.
7. Cada microservicio debe contar con pruebas unitarias para validar su funcionalidad de manera aislada, considerando el microservicio como una unidad independiente. Además, se deben implementar pruebas de integración que verifiquen su correcto funcionamiento en conjunto con otros microservicios.
8. Las pruebas de integración deben tener en cuenta las particularidades de los sistemas distribuidos, como la posible pérdida de mensajes, particiones de red o caídas de microservicios, para garantizar la robustez y la fiabilidad del sistema en su conjunto.

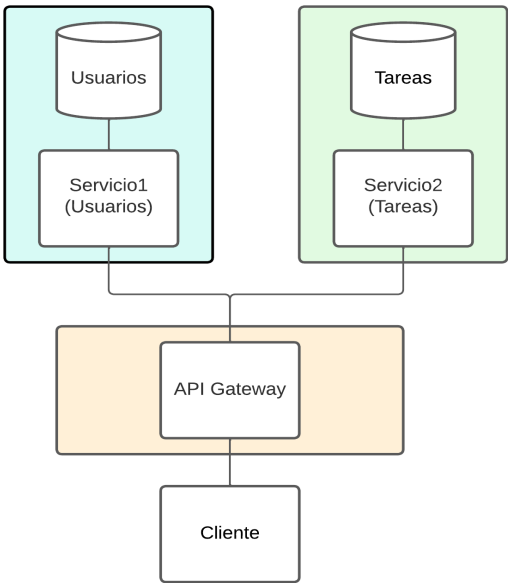

Fig. 3. Microservicio obtenido del desglose del monolito.

Siguiendo la guía previa para descomponer un monolito en microservicios, hemos obtenido un diseño que muestra claramente dos dominios distintos: uno enfocado en la gestión de tareas y otro en usuarios. En nuestro caso, hemos optado por mantener el lenguaje de programación Go (Golang) debido a su capacidad completa para construir cada servicio. Cambiar de lenguaje podría complicar las pruebas y comparaciones de rendimiento en el futuro.(**Fig**.3)

No hemos identificado dependencias significativas entre los módulos, lo que nos ha permitido evitar la interconexión entre ellos. Para aumentar la independencia y evitar

problemas de sincronización, dividimos la base de datos en dos partes. Posteriormente, empleamos un *API Gateway* para crear una interfaz común para todos los clientes, independientemente de los cambios en los microservicios, todo conectado a través de *Api Rest*.

3.2 **Despliegue del monolito**

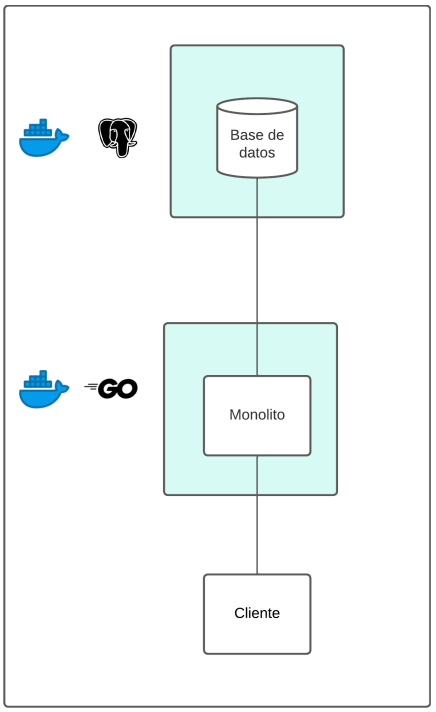

Fig. 4. Esquema de despliegue del monolito.

Pasos seguidos para el despliegue (**Fig**.4):

1. Montamos un contenedor con la imagen de postgres uno en el puerto 5432:5432.
2. Creamos la imagen de la aplicación en un contenedor en el puerto 3000:3000 verificando que se conecte correctamente con su base de datos respectiva.

3.3 **Despliegue del microservicio**

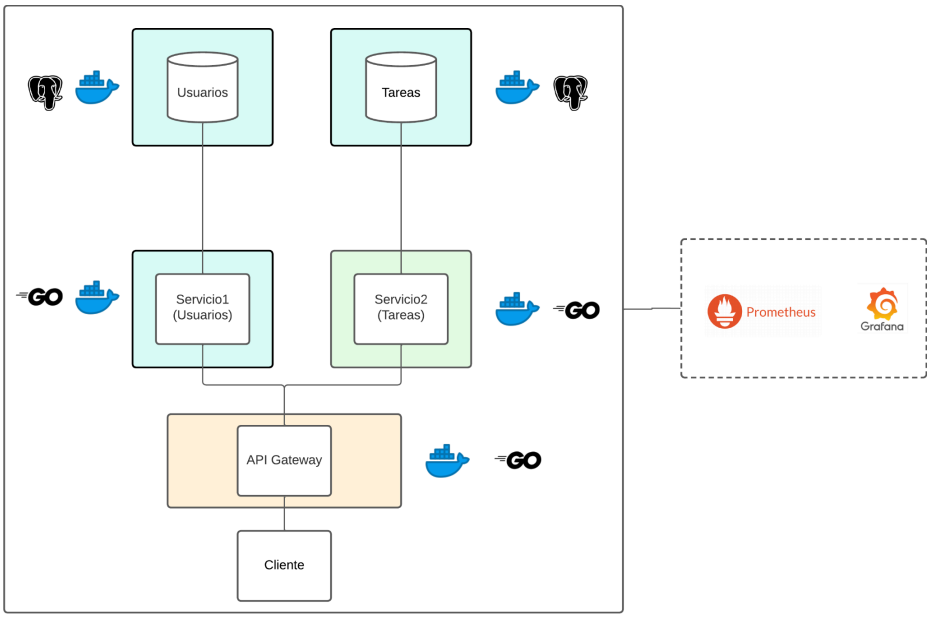

Fig. 5. Esquema de despliegue del microservicio.

Para el despliegue de la aplicación seguimos los siguientes pasos (**Fig**.5) :

1. Montamos dos contenedores con la imagen de *postgres* uno en el puerto 5432:5432 para la base de datos de tareas y otro en el puerto 5435:5432 para la base de datos de usuarios.
2. Creamos la imagen del microservicio de tareas y ejecutamos un contenedor con el puerto 4000:4000 verificando que se conecte correctamente con su base datos respectiva, y hacemos lo mismo con el microservicio de usuarios pero en el puerto 5000:5000.
3. Creamos una imagen del api gateway y lo hacemos correr el puerto 8080:8080.
4. Creamos dos *postgre exported* cada uno en un contenedor y enlazamos con a cada uno con su base de datos, con los puertos con los puertos 9187:9187 y 9188:9187.
5. Generamos un archivo prometheus.yml donde guardamos todas las fuentes de métricas entre estas los dos progress exported y el api gateway.
6. Corremos un contenedor docker con Prometheus usando la configuración anterior en el puerto 9090:9090.

El código y ejemplos de como montar el sistemas se encuentra en estos dos repositorios:

- https://github.com/RodrigoGonzalez78/microservicios_task_managment
- https://github.com/RodrigoGonzalez78/tasks_management_backend.git

4 Comparativa

Para la comparativa decidimos medir la el rendimiento de ambas arquitectura bajo el mismo sistema utilizando la herramienta jmeter y generando una prueba de estrés que se compone de cuatro pruebas de 30 segundos cada uno donde luego se promedia automáticamente por la herramienta, ejemplo pruebas de 2000 para */users* seria 4 pruebas de 2000 peticiones en 30 segundos todo esto para evitar anomalías e irregularidades en los tiempos de respuestas y en la tasa de errores(peticiones fallidas al servicio). (**Tabla**.1,2,3)

Tabla 1. Tabla de prueba de carga del Monolito.

Peticiones	servicio	Tiempo de respuesta máximo en milisegundos	Tiempo de Respuesta promedio en milisegundos	Tasa de error	Peticiones por segundo
10000	/users	1533	105	8.67%	216
	/task	1535	116	8.02%	216
5000	/users	2005	104	2.44%	141.9
	/task	2012	116	2.46%	141.8
2000	/users	1730	131	0.62%	53.4
	/task	1750	123	0.60%	53.4

Tabla 2. Tabla de prueba de carga del Microservicio sin *Api Gateway*.

Peticiones	servicio	Tiempo de respuesta máximo en milisegundos	Tiempo de Respuesta promedio en milisegundos	Tasa de error	Peticiones por segundo
10000	/users	17022	469	14.41%	222.4

Tabla 3. Tabla de la prueba de carga del Microservicio con *Api Gateway*.

Peticiones	servicio	Tiempo de respuesta máximo en milisegundos	Tiempo de Respuesta promedio en milisegundos	Tasa de error	Peticiones por segundo
10000	/users	11816	667	20%	241
	/task	41305	3773	13.25%	228
5000	/users	2010	212	4.05%	92.1
	/task	1916	202	3.19%	92.1
2000	/users	1242	204	0.0%	59
	/task	1417	188	0.0%	59.1

Consumo de CPU de cada arquitectura:

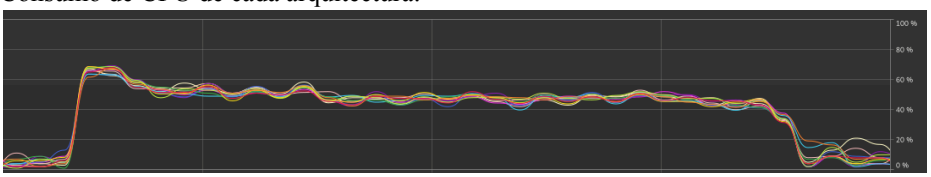

Fig. 6. Consumo de CPU del monolito durante la prueba.

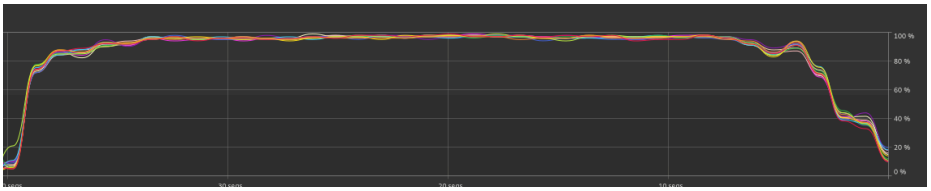

Fig. 7. Consumo de CPU del Microservicio durante la prueba.

Usamos como referencia el consumo de CPU como es el recurso donde se nota una mayor diferencia entre ambas arquitecturas, en esta comparación podemos ver un aumento significativo al usar una arquitectura de microservicio pasando de un 40-65% ver (**Fig. 6**) a un 100% ver (**Fig. 7**) igualmente esto puede variar dependiendo que función cumpla nuestro sistema y qué tecnologías usamos.

5 Conclusión y posibles mejoras

En esta investigación, se ha explorado la transición de un enfoque monolítico a una arquitectura de microservicios en una aplicación de gestión de tareas, documentando el paso a paso. Se ha seguido un enfoque meticuloso de diseño de una *API Rest* para cada dominio, permitiendo una clara separación de responsabilidades.
Los resultados de la experimentación, comparando el rendimiento del monolito con los microservicios, revelan diferencias significativas en términos de tiempos de respuesta, tasas de error y consumo de recursos. Mientras el monolito mostró tiempos de respuesta más rápidos y una menor tasa de error, y los microservicios presentaron tiempos de respuesta más prolongados y tasas de error más elevadas. Es interesante destacar que la presencia de un *API Gateway* parece influir positivamente en la eficiencia del microservicio. Estos hallazgos sugieren que la adopción de microservicios en una aplicación introduce complejidades adicionales en términos de rendimiento, pero ofrece ventajas en escalabilidad, mantenibilidad y flexibilidad en el desarrollo y despliegue de nuevas funcionalidades. Por lo tanto, al elegir una arquitectura para construir nuestros sistemas, debemos considerar si sacrificar el rendimiento y la sencillez que ofrecen los monolitos por la modularidad y la independencia que brindan los microservicios.

References

1. https://aws.amazon.com/es/microservices/, último acceso 2023.
2. https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419 , último acceso 2023.
3. https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#monolith_benefits , último acesso 2023.
4. https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#monolith_challenges, último acceso 2023.
5. https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices ,último acceso 2023.
6. https://www.atlassian.com/es/microservices/microservices-architecture/building-microservices, último acceso 2023.
7. https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#microservices-benefits, último acceso 2023.
8. https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#microservices_challenges, último acceso 2023.
9. https://www.hiberus.com/crecemos-contigo/migracion-microservicios/, último acceso 2023.

Definiciones

Integración continua (CI): Práctica de desarrollo de *software* que consiste en integrar y verificar el código fuente de manera automática y frecuente, para detectar errores y conflictos de manera temprana.

Implementación continua (CD): Práctica de desarrollo de *software* que consiste en automatizar el proceso de despliegue y entrega de *software*, para que los cambios se puedan implementar de manera rápida y segura.

Escalabilidad: Capacidad de un sistema para manejar un aumento en la carga de trabajo o el número de usuarios sin degradar su rendimiento o funcionalidad.

Cohesión: Grado en que los elementos de un módulo o componente están relacionados y trabajan juntos para lograr un objetivo común.

Acoplamiento: Grado en que los elementos de un módulo o componente dependen entre sí. Un acoplamiento alto significa que los cambios en un elemento pueden afectar a otros elementos.

API: Interfaz de programación de aplicaciones, conjunto de reglas y protocolos que permiten a los programas comunicarse entre sí.

API REST: Interfaz de programación de aplicaciones basada en el protocolo HTTP y los principios de la arquitectura REST (*Representational State Transfer*). Las API REST permiten a los clientes acceder y manipular recursos en un servidor a través de operaciones HTTP estándar, como GET, POST, PUT y DELETE. Las respuestas de la API REST suelen estar en formato JSON o XML.

Contenedores (Docker): Los contenedores son entornos aislados y portátiles que contienen todo lo necesario para que una aplicación se ejecute, incluyendo el código, las bibliotecas y las dependencias. Docker facilita la creación y gestión de entornos de desarrollo y producción, y permite una mayor flexibilidad y escalabilidad en el despliegue de aplicaciones.

Imagen en un contenedor: Plantilla de solo lectura que contiene todo lo necesario para ejecutar una aplicación en un contenedor de Docker. Las imágenes se crean a partir de un archivo de configuración llamado Dockerfile, que especifica las dependencias, bibliotecas y comandos necesarios para construir la imagen. Al crear un contenedor a partir de una imagen, se crea una instancia en ejecución de la aplicación con su propio sistema de archivos y recursos aislados.