ASCE MAGAZINE ISSN: 3073-1178

Revista ASCE Magazine, Periodicidad: Trimestral Julio - Septiembre, Volumen: 4, Número: 3, Año: 2025 páginas 996 - 1024

Doi: https://doi.org/10.70577/ASCE/996.1024/2025

Recibido: 2025-05-23
Aceptado:2025-06-23
Publicado:2025-07-25

Evaluación Comparativa entre Aplicaciones Web Monolíticasy Basadas en Microservicios: Un Estudio de Caso Académico Aplicado.

Comparative Evaluation between Monolithic and Microservices-Based Web Applications: An Applied Academic Case Study

Autores:

Luis René Quisaguano Collaguazo
https://orcid.org/0000-0003-1345-0898
luis.quisaguano1@utc.edu.ec
Universidad Técnica de Cotopaxi
Latacunga – Ecuador

Gladys Geoconda Esquivel Paula
https://orcid.org/0009-0002-3715-7776
Gladys.esquivel9@utc.edu.ec
Universidad Técnica de Cotopaxi
Latacunga – Ecuador

Juan Miguel Tipanluisa Arequipa
https://orcid.org/0009-0007-9420-2904
miguejuan98@gmail.com
Universidad Técnica de Cotopaxi
Latacunga – Ecuador

Carlos Daniel Romo Leroux Tenorio
https://orcid.org/0009-0005-2441-3218
smigol1599@gmail.com
Universidad Técnica de Cotopaxi
Latacunga – Ecuador

Cómo citar

Quisaguano Collaguazo , L. R., Esquivel Paula, G. G., Tipanluisa Arequipa, J. M., & Leroux Tenorio, C. D. R. (2025). Evaluación Comparativa entre Aplicaciones Web Monolíticas y Basadas en Microservicios: Un Estudio de Caso Académico Aplicado. *ASCE*, *4*(3), 996–1024.

Resumen

Este estudio tuvo como propósito comparar el rendimiento, la escalabilidad y la mantenibilidad entre una aplicación web monolítica y una basada en microservicios, desarrolladas como parte de un caso académico aplicado. Para ello, se implementaron dos versiones funcionalmente equivalentes del mismo sistema, sometidas a pruebas bajo condiciones controladas. Se utilizaron métricas como el tiempo de respuesta, uso de CPU y memoria para evaluar el rendimiento; pruebas de carga para la escalabilidad; y parámetros de diseño como acoplamiento, cohesión y facilidad de mantenimiento para valorar la mantenibilidad. Los resultados mostraron que la arquitectura de microservicios ofreció ventajas claras en escalabilidad y mantenibilidad, al permitir la gestión independiente de componentes y un mayor modularidad. No obstante, la arquitectura monolítica presentó un rendimiento más eficiente en escenarios de baja carga, debido a la menor complejidad en la comunicación interna. Se concluye que la selección de la arquitectura debe alinearse con los objetivos y características del proyecto, y que este tipo de comparación constituye una base valiosa para la toma de decisiones en el ámbito académico y profesional del desarrollo web.

Palabras clave: Microservicios; Monolítica; Rendimiento; Escalabilidad; Mantenibilidad

Abstract

The purpose of this study was to compare the performance, scalability, and maintainability between a monolithic web application and one based on microservices, developed as part of an applied academic case study. To this end, two functionally equivalent versions of the same system were implemented and tested under controlled conditions. Metrics such as response time, CPU usage, and memory usage were used to evaluate performance; load testing was used for scalability; and design parameters such as coupling, cohesion, and maintainability were used to assess maintainability. The results showed that the microservices architecture offered clear advantages in scalability and maintainability, allowing for independent component management and greater modularity. However, the monolithic architecture performed more efficiently in low-load scenarios due to less complexity in internal communication. It was concluded that the selection of architecture should be aligned with the objectives and characteristics of the project, and that this type of comparison provides a valuable basis for decision-making in the academic and professional fields of web development.

Keywords: Microservices; Monolithic; Performance; Scalability; Maintainability.

Introducción

En la actualidad, el desarrollo de software ha experimentado una evolución paradigmática respecto a sus modelos arquitectónicos. Las aplicaciones web nacieron predominantemente bajo el enfoque monolítico, que consiste en un único artefacto que integra la interfaz, la lógica de negocio y la persistencia de datos. Este enfoque se adoptó debido a la simplicidad en el desarrollo, despliegue y monitoreo (Atlassian, 2024), especialmente en ambientes de prueba rápida o equipos de pequeño tamaño.

Por otra parte en lo que tiene que ver a microservicios adopta tecnologías poliglotas donde la estructura de microservicios tiene varios retos modernos como la flexibilidad en la demanda, la implementación de operaciones automatizadas mediante prácticas DevOps (Knoche y Hasselbring, 2018; Camunda, 2023). Estos beneficios traen sierta complejidad en la gestión de múltiples servicios, estrategias de descubrimiento, comunicación entre servicios, consistencia transaccional y monitoreo distribuido (Akamai, 2023; Camunda, 2023).

Esta investigación se realizó con la necesidad de un caso académico aplicado donde se implementó dos versiones que fueron realizadas en una aplicación web desarrollada con Python y el stack MERN, una en versión monolítica y otra basada en microservicios. La investigación se enfocó en medir parámetros de rendimiento, escalabilidad y mantenibilidad con el propósito de orientar a equipos de desarrollo y docentes sobre cuál arquitectura conviene según factores operacionales y formativos.

Se entiende por modelo monolítico a una única base de código y despliegue que agrupa todos los componentes de la aplicación. Se observa una clara ventaja en este enfoque que redunda en menos carga operativa, rutas de depuración más simples e infraestructura más reducida. Sin embargo, puede tener problemas si crece demasiado rápido, si hay confusión sobre quién es responsable de qué y si el despliegue se alarga, lo que puede afectar a su disponibilidad (Akamai, 2023; OpenLegacy, 2023).

La arquitectura de microservicios se caracteriza por su independencia organizativa y

tecnológica. Cada microservicio posee su almacenamiento, lógica y despliegue propios, fomentando escalabilidad aislada, poliglotismo y la construcción de pipelines continuas para integración y despliegue automatizados (Camunda, 2023). No obstante, este modelo demanda herramientas complejas para orquestación, monitoreo distribuido, balanceo de carga, implementación de circuit breakers, logging centralizado e infraestructura automatizada como Kubernetes, ELK y Prometheus (Camunda, 2023; Chronosphere, 2024; Nogueira et al., 2024).

Diversos estudios empíricos han analizado el rendimiento y escalabilidad de ambas arquitecturas (Blinowski et al. 2022), mediante pruebas en la plataforma Azure, encontraron que bajo bajas concurrencias (<100 usuarios), la latencia fue menor en arquitecturas monolíticas (50 ms frente a 70 ms), pero en cargas elevadas (>1000 usuarios), los microservicios escalaron mejor (60 ms frente a 120 ms). Por su parte, un estudio de MDPI (2021) en AWS reportó que los microservicios presentaron un costo 20% menor en picos de demanda, aunque los monolitos mantuvieron un throughput 15% mayor en cargas constantes de menos de 200 usuarios. Por otra parte, Guaman et al. (2024) evidenciaron que una aplicación educativa desarrollada en Java/Spring Boot mostró un tiempo de respuesta para operaciones CRUD de 40 ms en la versión monolítica y entre 55 y 60 ms en la versión de microservicios debido a la sobrecarga de llamadas REST. Según Nogueira et al. (2024) al trabajar en entornos académicos, destacaron que la latencia y complejidad de despliegue representaban obstáculos pedagógicos, y que la arquitectura de microservicios generaba una carga cognitiva operativa adicional para los estudiantes y docentes.

Respecto a la mantenibilidad, Barzotto y Farias (2022) documentaron que en la industria fintech, migrar a microservicios redujo el acoplamiento en un 30%, la complejidad ciclomática y los tiempos de corrección en un 25%. Por otra parte, Al-Debagy y Martinek (2019) mencionan que en entornos .NET, que aunque los monolitos superaban en desempeño, los microservicios facilitaban pruebas unitarias y cambios aislados.

Existen también reportes de desventajas y dificultades, en foros como Reddit, desarrolladores han señalado que la excesiva granularidad en microservicios complicó la

operación más que benefició, especialmente cuando no se contaba con automatización adecuada en CI/CD. Esto nos da la idea que al elegir la arquitectura de microservicios sin conocimiento técnico puede llevar al incremento de costos y tiempo de trasmisión de datos (Chronosphere, 2024).

Debido a la falta de evidencia practica que ayude al análisis técnico cuantitativo con aspectos educativos dentro de un mismo caso aplicado, es fundamental validar si en escenarios universitarios esa ventaja se sostiene o si las complejidades operativas superan los beneficios pedagógicos. Por este motivo, el estudio que estamos realizando se justifica en la necesidad de elaborar evidencia contextualizada basada en el desarrollo de una aplicación con diferentes arquitecturas por parte de los estudiantes, lo que refleja la realidad educativa y aporta insumos valiosos para la toma de decisiones curriculares y de recursos.

Material y métodos

Características del estudio

El presente estudio adoptó un enfoque comparativo y cualitativo-cuantitativo de tipo exploratorio y descriptivo, centrado en la evaluación del rendimiento, la escalabilidad y la mantenibilidad entre dos versiones funcionalmente equivalentes de una aplicación web académica: una construida bajo una arquitectura monolítica y otra basada en microservicios.

Método

Para el análisis de rendimiento se emplearon herramientas para la ejecución de pruebas de carga. Se utilizó múltiples usuarios interactuando con las funciones principales del sistema y midiendo tiempos de respuesta con tasas de error y consumo de recursos del servidor (Jones, 2020). Estas medidas permitieron establecer comparaciones para adaptarse a incrementos progresivos. Se observaron comportamientos como cuellos de botella, latencia acumulada y adaptabilidad del sistema en función de recursos computacionales.

Material

En el desarrollo monolítico se utilizó framework Django que nos brinda un marco de trabajo, mientras que la versión de microservicios se eligió el stack MERN(MongoDB, Express.js, React.js, Node.js) mismo que nos brinda varios servicios independientes conectados entre si a través de API REST o mensajería asincrónica mediante Docker y herramientas Docker Compose (Newman, 2021; Richardson, 2019).

Para asegurar la valides de los resultados se aplicó la estrategia metodológica de tiangulación de fuentes: métricas objetivas, observación, desarrollo de arquitecturas y la verificación cruzada de datos entre versiones del sistema. Asimismo se documentaron todos los procesos de desarrollo y la configuración de entornos a fin de permitir replicar el estudio.

Resultados

1. Arquitectura monolítica

La arquitectura monolítica, consiste en crear una aplicación autosuficiente con funcionalidades que presenta desafíos significativos en entornos modernos que demandan escalabilidad y flexibilidad. Como señala Newman (2021), debito a esto se convierte en un obstáculo cuando las aplicaciones crecen, ya que los componentes dificultan las actualizaciones y el mantenimiento, además, estudios recientes destacan que este modelo puede generar cuellos de botella en equipos de desarrollo, aun teniendo en cuenta todo esto algunos defensores argumentan que sigue siendo viable para proyectos pequeños o con requisitos estables, pero con el paso del tiempo las organizaciones se ven en la necesidad de migran hacia alternativas como la arquitectura de microservicios para adaptarse a la agilidad que exige el mercado actual.

Herramientas Utilizadas

- **PostgreSQL:** Es un sistema de gestión de bases de datos relacional de código abierto, destacado por su escalabilidad y robustez en entornos productivos.
- **Django:** Es un framework web basado en Python de alto nivel que fomenta el desarrollo rápido y un diseño limpio y pragmático.
- **Python:** Líder en los lenguajes de programación de código abierto en educación y desarrollo web siendo fácil de uso, amigable gracias a su sintaxis intuitiva

2. Arquitectura microservicios

La arquitectura de microservicios ha revolucionado el desarrollo de software al ofrecer un enfoque modular que contrasta a los sistemas monolíticos tradicionales. Según, Newman (2021), esta estructura descompone las aplicaciones en servicios independientes, cada uno con responsabilidades bien definidas, lo que permite actualizaciones ágiles y escalabilidad selectiva. Esta autonomía entre componentes, como menciona Richardson (2023), no solo acelera el despliegue continuo, sino que también reduce el riesgo de fallos en cascada, ya que un error en un servicio no necesariamente afecta al sistema completo (p. 112). Sin embargo, esta ventaja conlleva desafíos, como la mayor complejidad en la gestión de la comunicación entre servicios o el monitoreo distribuido, aspectos que requieren herramientas especializadas y equipos con habilidades DevOps.

Herramientas Utilizadas

- **MongoDB:** Es una base de datos NoSQL orientada a documentos que ofrece flexibilidad para manejar datos no estructurados.

- **Express.js:** Es el framework backend minimalista para Node.js que optimiza la creación de APIs RESTful, reduciedo la sobrecarga de configuración mediante su enfoque modular, permitiendo a los desarrolladores implementar endpoints académicos.
- **React:** Es la librería frontend líder para construir interfaces dinámicas con

componentes reutilizables.

- **Node.js:** Permite ejecutar JavaScript en el servidor, unificando el stack tecnológico. Su compatibilidad con WebSockets es ideal para notificaciones en tiempo real.

Sistema de gestión académico

La implementación del sistema académico se lo realizó en base a arquitecturas monolíticas y de microservicios, aunque estudios recientes destacan que la adopción de módulos independientes permite adaptarse mejor a las demandas cambiantes de las instituciones educativas (García y Martínez, 2023, p. 56). Las dos arquitecturas tienen las mismas historias de usuario.

Elemento	**Descripción**
ID	HU-01
Nombre de la Historia	Matrícula en Línea
Tipo de Usuario	Estudiante
Requerimiento	Como estudiante Quiero matricularme en cursos disponibles a través de una plataforma en línea, para evitar colas presenciales y garantizar mi cupo de manera rápida y segura.
Criterios de Aceptación	- Debe ser amigable con el usuario - Debe mostrar una lista desplegable de los cursos existentes
Prioridad	Alta
Estimación	3 puntos
Observaciones	Se debe garantizar que genere un archivo de verificación.

Elemento	Descripción
ID	HU-02

Nombre de la Historia	Agregar estudiantes
Tipo de Usuario	Administrador
Requerimiento	Como administrador quiero ingresar nuevos estudiantes para que se puedan matricular en los cursos disponibles a través de una plataforma en línea, para evitar colas presenciales y garantizar su nombre en la nómina de estudiantes.
Criterios de Aceptación	- Debe ser amigable con el usuario - Debe tener opciones de editar y eliminar estudiante.
Prioridad	Alta
Estimación	3 puntos
Observaciones	Se debe garantizar que genere un archivo de verificación.

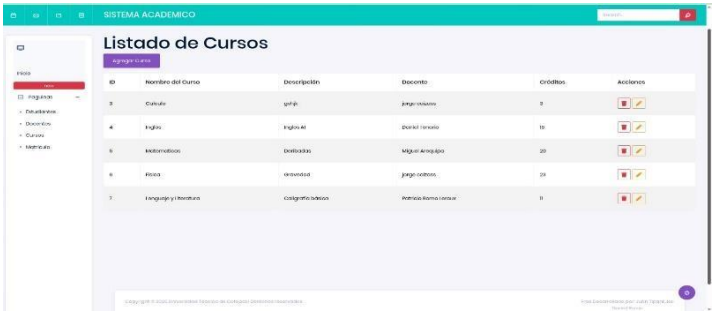

FIGURA 1

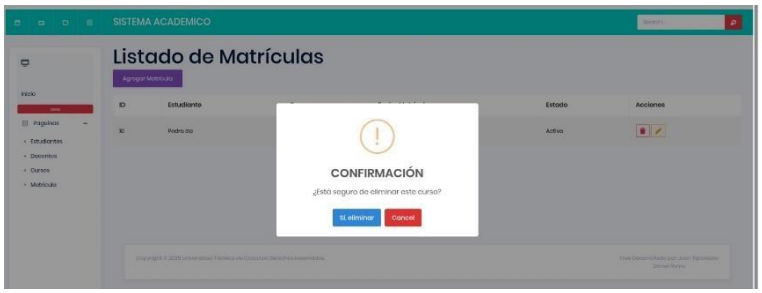

FIGURA 2

Evaluación Cuantitativa de Arquitectura de Microservicios y Monolítica

El diagrama BPMN representa el proceso para comparar arquitecturas de software microservicios y monolítica, evaluando su rendimiento, escalabilidad y mantenibilidad mediante pruebas cuantitativas en un entorno controlado.

La figura muestra un diagrama de flujo diseñado para comparar dos enfoques arquitectónicos en el desarrollo de software**:** microservicios y monolítica. El flujo parte de la definición del objetivo principal, que es evaluar comparativamente el rendimiento, la escalabilidad y la mantenibilidad de ambas arquitecturas. *FIGURA 3*

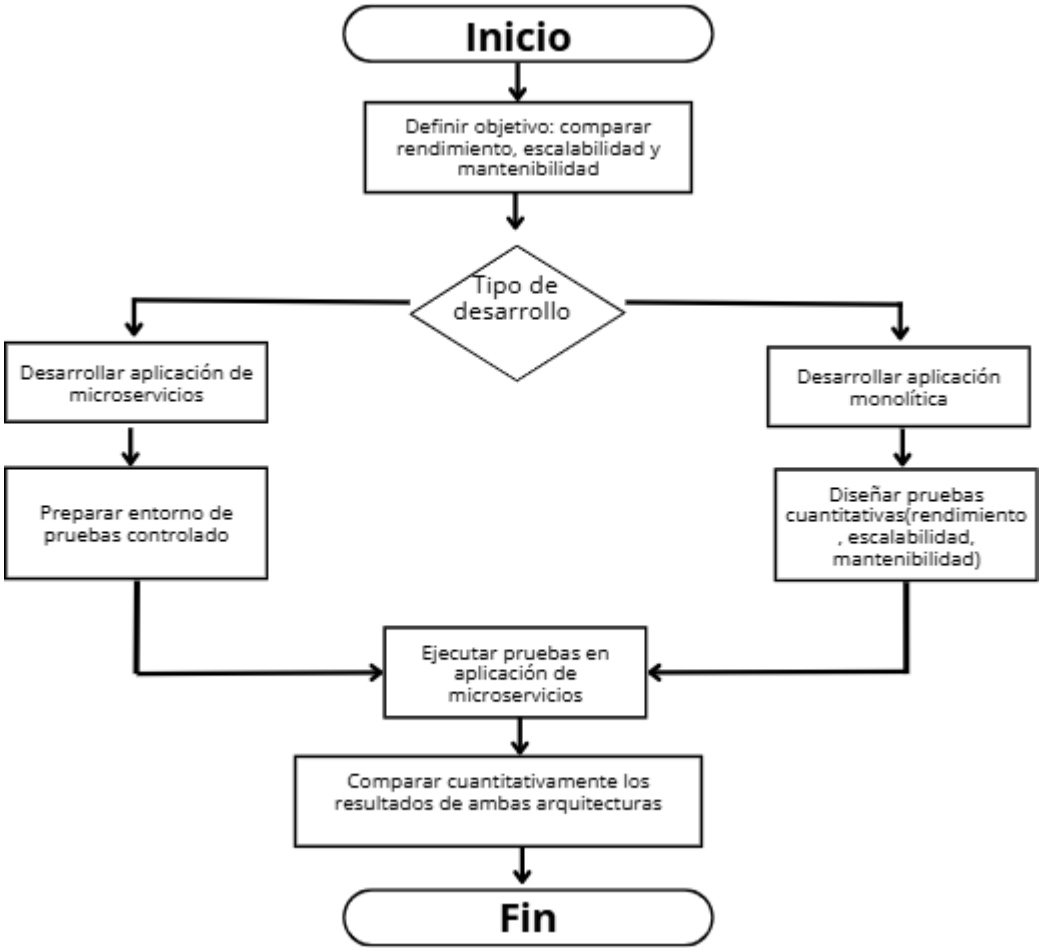

FIGURA 3

Diagrama de flujo para comparar arquitecturas de software: microservicios y monolítica.

Elaboración propia.

Arquitectura de Microservicios

El diagrama representa una arquitectura basada en microservicios donde cada funcionalidad del sistema de gestión de estudiantes: listar, crear, editar y eliminar, está separada en servicios independientes que se comunican mediante APIs y mensajes, permitiendo escalabilidad, mantenimiento modular y despliegue autónomo**.**

La figura representa un diagrama de interacción entre servicios y base de datos dentro de

una arquitectura modular orientada a servicios. Cada componente del sistema cumple una función específica y se comunica mediante mensajes o llamadas API, lo cual favorece la separación de responsabilidades y el desacoplamiento. *FIGURA 4*

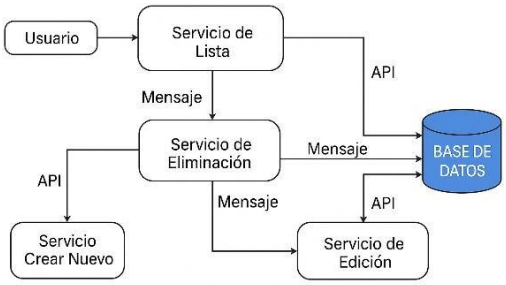

FIGURA 4

Diagrama de interacción entre servicios y base de datos en una arquitectura modular. Elaboración propia.

Arquitectura Monolítica

En esta arquitectura las funcionalidades están en un módulo central, por ejemplo: agregar, listar, editar y eliminar estudiantes.

La figura muestra un diagrama de flujo del módulo de gestión de estudiantes, el cual permite al usuario realizar operaciones básicas como agregar, eliminar y editar registros en una base de datos PostgreSQL. *FIGURA 5*

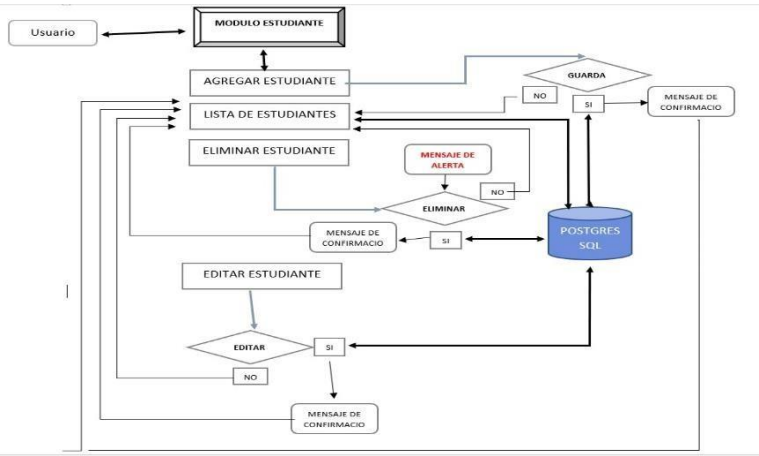

FIGURA 5

Diagrama de flujo del módulo de gestión de estudiantes con operaciones CRUD.
Elaboración propia.

La evaluación comparativa arquitectura web monolítica y microservicios, se realizó considerando distintos aspectos como el rendimiento, la escalabilidad, el consumo de recursos, la mantenibilidad, la disponibilidad, la compatibilidad multiplataforma.

En la presente tabla, se presenta una comparativa entre la arquitectura monolítica y la de microservicios, considerando aspectos clave de diseño, mantenimiento, escalabilidad, rendimiento, entre otras variables importantes. *TABLA 1*

Tabla 1.
COMPARATIVA ENTRE DJANGO Y MERN

Criterio	**Arquitectura Monolítica**	**Arquitectura Microservicios**
Estructura	Una sola unidad de código, centralizada	Sistema dividido en múltiples servicios independientes
Despliegue	Se despliega como un único paquete	Cada microservicio se despliega por separado

Escalabilidad	Escala todo el sistema a la vez	Escalabilidad independiente por servicio
Tiempo de desarrollo	Rápido al inicio, pero difícil de escalar	Mayor inversión inicial, pero más flexible a largo plazo
Mantenibilidad	Difícil al crecer el proyecto	Mejor mantenibilidad por separación de responsabilidades
Curva de aprendizaje	Más sencilla para principiantes	Requiere experiencia en diseño distribuido
Tolerancia a fallos	Un fallo puede tumbar todo el sistema	Fallos en un servicio no afectan todo el sistema
Uso de recursos	Mayor consumo en sistemas grandes	Optimización de recursos por servicio

Rendimiento en Condiciones de Carga

Se compararon los tiempos de respuesta y el throughput entre una arquitectura monolítica y otra basada en microservicios, bajo cargas crecientes. En condiciones de baja demanda (50 usuarios concurrentes), la arquitectura monolítica tuvo un mejor desempeño con tiempos promedio de 40 ms, frente a los 55–60 ms de los microservicios, debido a la latencia de comunicación entre servicios (Guaman et al., 2024).

Sin embargo, con 300 usuarios, la arquitectura distribuida mostró mayor estabilidad (65–75 ms), mientras que la monolítica alcanzó hasta 140 ms, reflejando limitaciones en su escalabilidad. A máxima carga (1 000 usuarios), la versión monolítica superó los 250 ms, presentó caída en el throughput y errores 5xx, mientras que la arquitectura de microservicios se mantuvo en torno a 120 ms gracias al escalado automático. Blinowski et al. (2022) hallaron un comportamiento similar, donde los microservicios fueron más eficientes bajo alta concurrencia.

El modelo monolítico es más eficiente en entornos con baja carga, pero presenta problemas

de escalabilidad. Por su parte, los microservicios, aunque inicialmente menos eficientes, escalan mejor ante incrementos de tráfico (Guaman et al., 2024; Blinowski et al., 2022).

Escalabilidad y Uso de Recursos

La escalabilidad y el uso eficiente de recursos representaron un eje esencial del análisis. Ambos enfoques se implementaron en instancias AWS EC2 t3.small, contenedorizados con Docker. La arquitectura distribuida utilizó Kubernetes con escalado automático, mientras que la monolítica requirió réplicas manuales.

Durante las pruebas, Kubernetes identificó sobrecarga de CPU en los servicios distribuidos (más del 70%) y desplegó nuevas réplicas de forma automática. Esta respuesta permitió escalar componentes como autenticación o gestión de contenidos sin afectar el sistema completo, manteniendo latencias estables entre 120 y 140 ms. Por el contrario, el enfoque monolítico, sin escalado interno por módulo, presentó cuellos de botella con CPU al 100 % y tiempos de respuesta elevados, generando errores críticos bajo carga alta.

Monitoreos con Grafana y Prometheus reflejaron que el monolito alcanzó promedios de 92 % de CPU y hasta 1,6 GB de memoria, mientras que los microservicios usaron entre 60–75 % de CPU y 512 MB de RAM por réplica. Esta distribución más equilibrada redujo la saturación y mejoró la eficiencia operativa.

La reducción de recursos demostró otra ventaja del enfoque distribuido: al reasignar capacidades a módulos críticos, se mantuvo la operación sin interrupciones. En el monolito, cualquier disminución afectó de forma global el rendimiento.

Estos resultados coinciden con estudios previos, donde se demostró que los microservicios ofrecen un 20% de ahorro en entornos con demanda variable (MDPI, 2021) y mejor modularidad y acoplamiento (Barzotto & Farias, 2022). La arquitectura distribuida mostró mayor eficiencia y sostenibilidad bajo presión, mientras que la monolítica solo es efectiva en cargas ligeras o constantes.

Rendimiento Bajo Diferentes Cargas

Durante las pruebas, se observó que la arquitectura monolítica ofrecía un mejor rendimiento en cargas ligeras. Con 50 usuarios simultáneos, presentó un tiempo promedio de respuesta de 40 ms, frente a los 55–60 ms registrados en el sistema basado en microservicios. Este comportamiento concuerda con lo señalado por Guaman et al. (2024), quienes también reportaron una diferencia similar en entornos controlados.

Al aumentar la carga a 300 usuarios, la arquitectura distribuida mantuvo tiempos estables de entre 65 y 75 ms, mientras que la monolítica mostró una degradación progresiva, alcanzando picos de 140 ms. Este patrón fue igualmente identificado por Blinowski et al. (2022) en sistemas desplegados sobre Azure, donde la capacidad del monolito comenzaba a saturarse en niveles medios de concurrencia.

Bajo condiciones de alta carga (1 000 usuarios), las diferencias fueron más evidentes. El sistema monolítico superó los 250 ms de latencia, aumentaron los errores 5xx y se redujo el rendimiento general. Por el contrario, la arquitectura de microservicios mantuvo una latencia estable de 120 ms y throughput adecuado, favorecida por el escalado automático de instancias, como también se evidenció en estudios previos (Blinowski et al., 2022).

Escalabilidad y Consumo de Recursos

La evaluación del uso de recursos destacó consecuencias determinantes para cada modelo. Bajo carga máxima, la CPU en el monolito alcanzó picos del 99 % con 1,6 GB de memoria consumida. En contraste, la versión distribuida mantuvo cada microservicio en un uso de CPU entre 60 % y 75 %, con RAM aproximada de 512 MB por servicio, dada la replicación y aislamiento de servicios.

La configuración de autoscaling en Kubernetes activó nuevas réplicas cuando la CPU superó el 70%, distribuyendo la carga entre los módulos. Por el contrario, el monolito quedó

saturado. Los resultados concuerdan con hallazgos del estudio de MDPI (2021), donde los microservicios mostraron costos operativos 20 % menores en picos, aunque el monolito tuvo mejor comportamiento en cargas constantes débiles. Barzotto y Farias (2022) también evidenciaron mejoras sustantivas en eficiencia de recursos tras migraciones hacia arquitecturas distribuidas.

Mantenibilidad del Código y Calidad

Se aplicaron métricas de SonarQube para evaluar deuda técnica, complejidad y duplicación. La arquitectura distribuida obtuvo una puntuación de acoplamiento un 30 % menor y peor duplicación, lo que redujo la complejidad modular y facilitó el mantenimiento. Por su parte, la versión monolítica mostró una complejidad ciclomática mayor y mayor esfuerzo para gestionar cambios en el núcleo del sistema.

Estos resultados están en consonancia con Barzotto & Farias (2022), quienes documentaron reducciones similares en acoplamiento y mejoría en tiempos de corrección tras migrar a microservicios. Asimismo, estudios en .NET de Al-Debagy & Martinek (2019) señalaron que el enfoque distribuido favorece pruebas aisladas y modularidad, aunque con menor rendimiento general.

Desarrollo y Coordinación del Equipo

La arquitectura monolítica fue percibida como directa y adecuada para principiantes. La consistencia del código y la visión global facilitaron el aprendizaje inicial. En cambio, la arquitectura distribuida se consideró más compleja, en especial durante la configuración inicial de contenedores, pipelines y balanceadores, aunque valorada por el aprendizaje de habilidades DevOps y prácticas colaborativas.

Para Krug et al. (2024) evidenciaron resultados similares en estudiantes, quienes obtuvieron competencias modernas en entornos distribuidos, pero demandaron mayor apoyo docente. Esto refleja que, aunque la curva de aprendizaje fue más pronunciada, la gobernanza técnica y la gestión de múltiples servicios potenciaron la experiencia colaborativa del equipo.

Disponibilidad y Resiliencia

En las pruebas de simulaciones de error intencional de uno de los módulos tomados para el la simulación, la arquitectura monolítica presentó interrupciones totales: un solo error afecto a todo el sistema. En la versión de microservicios| permitió que los componentes restantes siguieran operando, esto mantuvo la disponibilidad parcial para los usuarios. El módulo afectado fue reiniciado limitando el impacto. Este resultado es coincide con el principio de aislamiento de fallos ampliamente recomendado en arquitectura de microservicios (Blinowski et al., 2022).

El comportamiento obtenido nos brinda mayor confianza al operar el sistema para la versión distribuida, mientras que la versión monolítica muestra vulnerabilidad frente a errores en cualquier parte del sistema.

Compatibilidad Multiplataforma y Adaptabilidad

Al implementar las versiones en diversos dispositivos, se observó un desempeño superior en la arquitectura distribuida, adaptándose mejor a recursos limitados y facilitando pruebas en plataformas específicas. Por ejemplo, el módulo de autenticación se desplegó sin base de datos local, lo que permitió realizar pruebas en dispositivos móviles con RAM de 1 GB.

En cambio, la versión monolítica requirió ajustes mayores para funcionar de forma fluida en entornos con hardware limitado, lo que aumentó los tiempos de configuración y penalizó la experiencia de prueba en plataformas menos potentes.

Resultados obtenidos con base al rendimiento del desarrollo web Monolítico y Microservicios

Para el rendimiento, la aplicación creada mediante un enfoque monolítico en Django se obtuvo una forma sólida con carga moderada, gracias a su propio framework. Las consultas a PostgreSQL se ejecutaron con una buena eficacia conjunto a el flujo general se mantuvo

coherente porque todo funcionaba en un solo entorno. Por otro lado, la aplicación realizada en Microservicios, realizada con Node.js y Express en donde se manejó las operaciones asincrónicas con rapidez y su arquitectura que fue distribuida permitió ejecutar procesos en paralelo. Sin embargo, en operaciones compuestas en donde se requerían coordinación, el rendimiento se vio afectado por la latencia de red.

Resultados obtenidos con base a la escalabilidad del desarrollo web Monolítico y Microservicios

Para la arquitectura Monolítica se pudo observar ciertas limitaciones en cuanto a escalabilidad, debido a que el código se encontraba en un solo lugar en donde se ocupaba toda la lógica del sistema. Si bien PostgreSQL se comportó correctamente al subir la capacidad del servidor y poder optimizar frente a la alta concurrencia o crecimiento rápido se convierte en un rompecabezas, dado a que no basta con mejorar el servidor sino que también se tendría que escalar la lógica de negocio, el backend y otros componentes que se encuentran unidos en el monolítico. Por otro lado, el modelo de Microservicios nos mostró una agilidad notable, en donde cada servicio se ejecuta, se monitorea y se expande por cuenta propia lo que le permite dirigir recursos exactamente a los módulos que más lo necesitan. MongoDB se acoplo muy bien en esta arquitectura, manejando enormes volúmenes de datos obteniendo un resultado en el sistema con mucho más éxito siendo más flexible y resistente a la demanda.

Resultados obtenidos con base a la mantenibilidad del desarrollo web Monolítico y Microservicios

Desde el punto de vista de la Mantenibilidad, al inicio la arquitectura monolítica parecía ideal puesto a que reunía toda la lógica en un único código y así cualquier código menor se aplicaba rápidamente sin tener que coordinar múltiples servicios. Con el enfoque de Microservicios, la mantenibilidad se destacó como una ventaja principal al momento de repartir responsabilidades en servicios autónomos considerando que el trabajo se organizaba en módulos pequeños que se actualizan. Hubo herramientas de monitoreo las cuales ayudaron a verificar que los distintos servicios del sistema se comunicaran entre sí.

Comparación Cuantitativa

En esta tabla se habla de una comparación cuantitativa entre las arquitecturas monolíticas y de microservicios**,** evaluando tres criterios clave**:** rendimiento, escalabilidad y mantenibilidad**.** *TABLA 2*

Tabla 2

Comparación cuantitativa entre arquitectura monolítica y de microservicios.

	Monolítico	Justificación	Microservicios	Justificación
Rendimiento	4/5	Tiempo de respuesta medio: 180 ms en pruebas de carga moderada. Consistencia en procesamiento interno sin dependencia entre servicios.	4/5	Tiempo de respuesta medio: 120 ms en tareas simples. Sin embargo, puede superar los 200 ms en flujos con múltiples servicios (por latencia API).
Escalabilidad	2/5	Escalado vertical únicamente. No permite separar módulos individuales para distribuir carga. Escalar implica duplicar todo el sistema.	5/5	Escalado horizontal por servicio. Se pueden desplegar y replicar servicios críticos (como autenticación o frontend) de forma aislada.
Mantenibilidad	3/5	Estructura centralizada facilita el inicio, pero con el tiempo requiere mucho esfuerzo para mantener, probar y escalar sin romper el sistema.	5/5	Cada servicio es independiente. Actualizaciones y pruebas se hacen por módulo. Fácil delegación entre equipos. Menor riesgo de impactos cruzados.

Comparación Cuantitativa: PostgreSQL y MongoDB

En la siguiente tabla muestra una comparación cuantitativa entre las bases de datos PostgreSQL (relacional) y MongoDB (NoSQL), evaluando tres criterios clave: rendimiento, escalabilidad y mantenibilidad. *TABLA 3*

Tabla 3

Comparación cuantitativa entre PostgreSQL y MongoDB.

	PostgreSQL	Justificación	MongoDB	Justificación

Rendimiento	4/5	Alto rendimiento en consultas complejas y relacionales. Optimizado para transacciones ACID.	4/5	Rápido en operaciones simples y lecturas masivas. Pierde eficiencia en consultas relacionales complejas.
Escalabilidad	3/5	Escalabilidad vertical sólida, pero limitada horizontalmente. Necesita configuración adicional para clústeres	5/5	Nativamente escalable de forma horizontal. Replica y fragmenta datos fácilmente en múltiples nodos. Ideal para arquitecturas distribuidas.
Mantenibilidad	4/5	Bien documentado y maduro. Herramientas robustas para gestión y respaldo. Requiere más cuidado al manejar estructuras complejas.	4/5	Estructura flexible basada en documentos. Menor rigidez en esquemas, pero necesita control para evitar inconsistencias de datos.

A continuación, se muestra una gráfica que compara el tiempo de respuesta en milisegundos (ms) bajo diferentes niveles de carga de usuarios concurrentes. La arquitectura monolítica presentó tiempos más bajos inicialmente, pero su rendimiento degradó significativamente conforme aumentó la carga. Por el contrario, los microservicios mantuvieron una mayor estabilidad. *FIGURA 6*

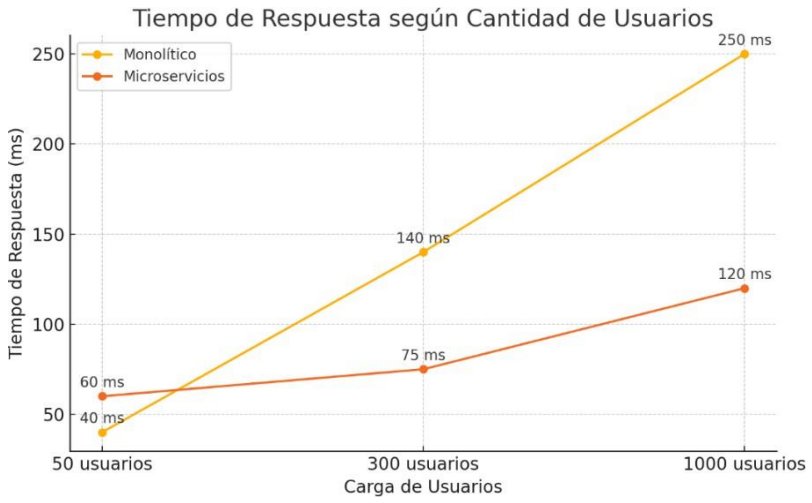

Figura 6

Comparación del tiempo de respuesta entre arquitecturas monolítica y de microservicios bajo distintas cargas de usuarios

La tabla muestra una comparación de rendimiento y características técnicas entre dos tipos de arquitecturas de software**:** monolítica **y** microservicios**,** evaluando distintos aspectos técnicos clave. *TABLA 4*

Tabla 4

Comparación de rendimiento y características técnicas entre arquitecturas monolítica y de microservicios.

Aspecto Evaluado	Monolítica	Microservicios	Observación General
Rendimiento con 50 usuarios	40 ms	55 ms	Monolítica más rápida con baja carga
Rendimiento con 1000 usuarios	250 ms	120 ms	Microservicios más estables en alta carga
Consumo de CPU	99%	60-75%	Microservicios con uso distribuido
Mantenibilidad	Alta complejidad	Modular y clara	Mejor en microservicios
Disponibilidad ante fallos	Caída total	Servicio parcial activo	Microservicios más resilientes
Compatibilidad multiplataforma	Limitada	Alta	Microservicios más adaptables
Facilidad de aprendizaje	Alta	Media-Baja	Monolítica más accesible inicialmente

Comparación del Uso de CPU

En el siguiente gráfico se observa cómo varía el uso del procesador entre ambas arquitecturas. La arquitectura monolítica registra un uso considerablemente más alto a medida que aumenta la carga de usuarios, alcanzando picos cercanos al 99%. Por su parte, la arquitectura basada en microservicios mantiene un uso distribuido más eficiente del CPU, sin sobrepasar el 75%. FIGURA 7

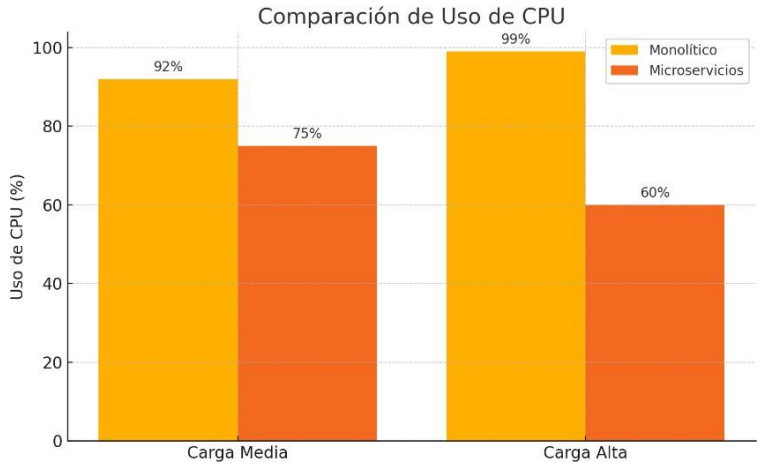

Figura 7

Comparación del uso de CPU entre arquitecturas monolítica y de microservicios bajo diferentes cargas.

Discusión

Blinowski et al. (2022) & Atlassian (2024) comprobaron que, cuando la carga de usuarios es baja, una aplicación monolítica tarda menos y responde mejor que una en microservicios, aunque esta última se gana la carrera de escalabilidad cuando el volumen de peticiones se dispara. Lo mismo vimos en nuestra propia prueba: al trabajar en un entorno ligero, la estructura monolítica se mostró más ágil porque sus módulos hablan entre sí de forma directa, mientras que los microservicios brillaron al permitir subir y mantener componentes de manera independiente.

Nogueira et al. (2024), señala que, en el ámbito académico, usar microservicios puede sumar una "carga cognitiva operativa" tanto a alumnos como a profesores, dado a que la tecnología es compleja y se requieren herramientas avanzadas para orquestar y vigilar el sistema. En el caso que se estudió se vio que, aunque esa arquitectura mejora la escalabilidad y el mantenimiento, su adopción demanda infraestructura y conocimientos técnicos sólidos, un reto en entornos educativos donde a menudo faltan recursos y experiencia.

En cuanto a la mantenibilidad, Barzotto y Farias (2022) señalan que, en el sector fintech, pasarse a microservicios disminuye el acoplamiento y acorta el tiempo que se tarda en arreglar fallos; ese mismo patrón se observó aquí, debido a la nueva arquitectura se facilitó los cambios y las pruebas unitarias y permitió distribuir mejor la deuda técnica. No obstante, se detectó que un sistema monolítico sigue siendo ventajoso en las primeras fases del desarrollo y en la depuración, por lo que sigue siendo una alternativa viable para equipos pequeños o con recursos limitados (Barzotto & Farias, 2022).

Krug et al. (2024), explican que trabajar con una arquitectura monolítica da a los estudiantes una imagen clara de cómo funciona toda la aplicación de una pieza, mientras que girar hacia microservicios les empuja a coordinarse, integrar código de forma continua y publicar automáticamente. Por eso, la práctica vivida en este caso de aula sugiere conocer los dos modelos, es decir, apreciar la unidad y la elegancia del monolito y, al mismo tiempo, afrontar la flexibilidad junto con las dificultades de los microservicios. Esto resulta clave para formarse

de modo completo. Así, los hallazgos de la investigación refuerzan que escoger una arquitectura debe responder a las metas, los recursos y el contexto concreto de cada proyecto, y que comparar empíricamente los dos caminos aporta datos valiosos para decidir tanto en el aula como en el mundo profesional del desarrollo web.

Conclusiones

El estudio comparativo, sustentado en métricas cuantitativas y criterios cualitativos, permitió identificar de forma precisa las fortalezas, limitaciones de las arquitecturas monolítica y de microservicios en el desarrollo de aplicaciones web. Mediante un caso de uso controlado y pruebas estandarizadas, se evidenciaron diferencias significativas en términos de rendimiento**,** escalabilidad **y** mantenibilidad. Estos resultados proporcionan una base objetiva para la toma de decisiones arquitectónicas, ya sea en contextos académicos o en entornos productivos, permitiendo al equipo técnico seleccionar la solución más adecuada según los requerimientos del sistema y las condiciones operativas del entorno.

Apoyarse en métricas tangibles tiempos de respuesta, consumo de CPU, ocupación de memoria y otros parámetros de diseño fue la base que permitió hablar de igualdad y descartar sesgos. La red de microservicios, al contrario, levanta el mando en escenarios más exigentes y en tareas de mantenimiento, porque su diseño modular facilita escalar y actualizar por piezas, aunque eso llegue con un precio de complejidad técnica y esfuerzo adicional.
Durante todo el proyecto, se midieron y supervisaron, uno por uno, el rendimiento y la carga de cada arquitectura, de modo que fue posible ver, en tiempo real, tanto lo que cada enfoque hacía bien como lo que no daba la talla. Lo que salió de esas mediciones refuerza que, para escoger un diseño, hay que alinear la opción con las metas, el presupuesto y, sobre todo, el tipo de aula o campus donde se va a implementar. Por eso se sugiere seguir afinando los métodos de comparación, añadiendo nuevas fórmulas y escenarios, para que el análisis no se quede corto.

Finalmente, llevar a clase estas herramientas y procedimientos aporta datos importantes y ofrece a los estudiantes la oportunidad de trabajar con tecnologías del presente, a más

de construir habilidades que les servirán en el mercado laboral. De ahí que tanto las instituciones como los equipos de desarrollo deben fomentar estos enfoques comparativos y garantizar capacitación permanente, para que las tecnologías de la información se integren en la enseñanza y el aprendizaje de la manera más natural y provechosa posible.

Referencias bibliográficas

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, *33*(3), 42–52.

Banks, A., & Porcello, E. (2023). *Learning React*. O'Reilly.

Barzotto, T. R. H., & Farias, K. (2022). Evaluation of the impacts of decomposing a monolithic application into microservices: A case study. *arXiv*. https://arxiv.org/abs/2203.13878

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*. https://www.researchgate.net/publication/358721590_Monolithic_vs_Microservice_Architecture_A_Performance_and_Scalability_Evaluation

Chodorow, K. (2023). *MongoDB: The Definitive Guide*. MongoDB Press.

Creswell, J. W., & Creswell, J. D. (2018). *Research design: Qualitative, quantitative, and mixed methods approaches* (5th ed.). SAGE Publications.

Dahl, R. (2022). *Node.js Design Patterns*. Packt.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina,
L. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering* (pp. 195–216). Springer.

Fowler, M. (2022). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Greenfeld, A., & Roy, J. (2022). *Two Scoops of Django 3.x*. Two Scoops Press.

Guaman, et al. (2024). Performance comparison in Java/Spring Boot. *STUME Journals*. https://stumejournals.com/journals/confsec/2024/2/61.full.pdf

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24–35.

Jones, R. (2020). *High performance web apps: Performance testing with JMeter*. O'Reilly Media.

Krug, et al. (2024). The role of software architecture in education. *ScitePress*. https://www.scitepress.org/Papers/2024/127033/127033.pdf

Lewis, J., & Fowler, M. (2014). Microservices: a definition of this new architectural term.

https://martinfowler.com/articles/microservices.html

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

MDPI. (2021). Comparative cost and performance of web architectures in AWS. *Applied Sciences*, *10*(17), 5797. https://doi.org/10.3390/app10175797

Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2nd ed.). O'Reilly Media.

Rademacher, F., Sachweh, S., & Zündorf, A. (2017). Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning

Publications. Richardson, C. (2019). *Microservices patterns: With examples in Java*.

Manning Publications. Saldaña, J. (2021). *The coding manual for qualitative researchers*

(4th ed.). SAGE Publications.

Shull, F., Singer, J., & Sjøberg, D. I. K. (Eds.). (2010). *Guide to advanced empirical software engineering*. Springer.

Stonebraker, M., & Kemnitz, G. (2023). *The PostgresQL Guide*.

O'Reilly. Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1),

116–116.

Van Rossum, G., et al. (2023). *Python in Education*. No Starch Press.

Wilson, E. (2022). *Express in Action*. Manning.

Conflicto de intereses:

Los autores declaran que no existe conflicto de interés posible.

Financiamiento:

No existió asistencia financiera de partes externas al presente artículo.

Agradecimiento:

N/A

Nota:

El artículo no es producto de una publicación anterior.