UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

MAESTRÍA EN

SOFTWARE CON MENCIÓN EN DISEÑO DE ARQUITECTURA DE SISTEMAS

RPC-SO-34-NO.778-2021

OPCIÓN DE TITULACIÓN:

PROYECTO DE TITULACIÓN CON COMPONENTES DE INVESTIGACIÓN APLICADA Y/O DE DESARROLLO

TEMA:

DISEÑO DE UNA ARQUITECTURA EN MICROSERVICIOS CON INDEXACIÓN DINÁMICA Y DISTRIBUIDA PARA OPTIMIZAR LA BÚSQUEDA EN TIEMPO REAL DE LA DISPONIBILIDAD EN UN SISTEMA DE RESERVAS EMPRESARIAL COEXISTENTE CON UNA ARQUITECTURA MONOLÍTICA

AUTOR(ES)

LUIS FERNANDO MERINO SAQUICELA

DIRECTOR:

DANIEL GIOVANNY DÍAZ ORTIZ

QUITO – ECUADOR

2025

Autor(es):

Luis Fernando Merino Saquicela
Ingeniero de Sistemas
Candidato a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
lmerino@est.ups.edu.ec

Dirigido por:

Daniel Giovanny Díaz Ortiz
Ingeniero de Sistemas
Master Universitario en Software Libre, Master en Redes de Comunicaciones
ddiaz@ups.edu.ec

Luis Fernando Merino Saquicela
DISEÑO DE UNA ARQUITECTURA EN MICROSERVICIOS CON INDEXACIÓN DINÁMICA Y DISTRIBUIDA PARA OPTIMIZAR LA BÚSQUEDA EN TIEMPO REAL DE LA DISPONIBILIDAD EN UN SISTEMA DE RESERVAS EMPRESARIAL COEXISTENTE CON UNA ARQUITECTURA MONOLÍTICA

DEDICATORIA

"Mira hijo, la mejor herencia que deja un padre es la educación. Piensa, estudia, edúcate y triunfarás en la vida."

Palabras sabias de mi madre, que hoy resuenan en cada paso de mi camino. Hoy, más que nunca, comprendo la profundidad de su mensaje: el verdadero legado que un padre puede dejar no reside en lo material, sino en los valores, el conocimiento y el ejemplo.

Dedico este trabajo a la memoria de mis padres, Martha Lucia Saquicela Morocho y Euclides Neptalí Merino Manjarres. Su ejemplo, dedicación y profunda fe en la educación como motor de cambio sembraron en mí el compromiso de crecer, superar obstáculos y perseguir mis metas con responsabilidad y perseverancia

Este logro es una expresión viva de esa herencia invaluable que me dejaron, y una forma de honrar su memoria desde la gratitud y el compromiso con lo que me enseñaron.

AGRADECIMIENTO

A Pablo Enrique Saquicela Morocho y Luz de la Mercedes Richards León, junto a su familia, por haber sido testigos y cómplices de muchos momentos decisivos en mi vida. En los días de incertidumbre, su apoyo fue un refugio; en los de alegría, su presencia, una celebración. Gracias por su hospitalidad, por confiar en mí y por ofrecerme siempre una palabra sabia cuando más la precisaba. Su hogar ha sido, en muchos sentidos, una extensión del mío, y ellos una figura de padre y madre que Dios, con amor y sabiduría, puso en mi camino para orientarme cuando más lo necesitaba.

A toda mi familia, porque detrás de cada meta cumplida hay un pilar invisible pero firme, hecho de amor, sacrificio y entrega. Gracias por su fortaleza, por su fe en mí incluso en los momentos más difíciles, y por recordarme que detrás de cada paso hay siempre manos invisibles que sostienen, alientan y nunca sueltan.

Y a mi tutor, Daniel Giovanny Díaz Ortiz, por su orientación precisa, su criterio formativo y su compromiso durante todo este proceso. Su guía no solo fue académica, sino también humana, orientándome a abordar cada etapa del proceso investigativo con rigurosidad y responsabilidad académica, fiel al espíritu salesiano de formar buenos cristianos y honrados ciudadanos.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Diseño de una Arquitectura en Microservicios con Indexación Dinámica y Distribuida para Optimizar la Búsqueda en Tiempo Real de la Disponibilidad en un Sistema de Reservas Empresarial Coexistente con una Arquitectura Monolítica

Autor(es):

LUIS FERNANDO MERINO SAQUICELA

Resumen

El presente proyecto aborda el diseño de una arquitectura basada en microservicios con indexación dinámica y distribuida, orientada a optimizar la búsqueda en tiempo real de disponibilidad en un sistema empresarial de reservas que coexiste con una arquitectura monolítica. Frente a las limitaciones de escalabilidad y rendimiento identificadas en esta última, el estudio plantea una solución técnica mediante la adopción progresiva del patrón Strangler, integrando tecnologías especializadas como Elasticsearch para la indexación distribuida, Redis para el almacenamiento en caché y RabbitMQ para la sincronización asíncrona de datos.

Los resultados experimentales evidenciaron mejoras significativas en términos de latencia y procesamiento. En condiciones de alto tráfico, la versión monolítica original registró tiempos promedio de respuesta superiores a los 30 segundos. Con la integración de Elasticsearch, este tiempo se redujo a menos de 100 milisegundos en el sistema optimizado, evidenciando una mejora drástica en la eficiencia de las consultas. Asimismo, la integración de Redis contribuyó a estabilizar los tiempos de respuesta ante consultas recurrentes en contextos de alta concurrencia.

La validación técnica demostró que la arquitectura diseñada no solo cumple con los requerimientos de escalabilidad y rendimiento, sino que también sienta las bases para una evolución sostenible del sistema monolítico, facilitada por la modularidad intrínseca de los microservicios. La investigación aportó conocimientos valiosos sobre cómo las metodologías ágiles y la implementación estratégica de microservicios permiten transformar arquitecturas empresariales legadas, promoviendo entornos tecnológicos más flexibles, eficientes y alineados con las exigencias dinámicas del sector.

Palabras clave:
Microservicios, Indexación distribuida, Transformación arquitectónica, Modernización de sistemas legados, Búsqueda en tiempo real, Alta concurrencia.

Abstract

This project presents the design of a microservices-based architecture with dynamic and distributed indexing, aimed at optimizing real-time availability search in an enterprise reservation system that coexists with a monolithic architecture. To address the limitations of scalability and performance identified in the latter, the study proposes a technical solution through the progressive adoption of the Strangler pattern, integrating specialized technologies such as Elasticsearch for distributed indexing, Redis for caching, and RabbitMQ for asynchronous data synchronization.

Experimental results showed significant improvements in latency and processing capacity. Under high-traffic conditions, the original monolithic version recorded average response times exceeding 30 seconds. With the integration of Elasticsearch, this time was reduced to under 100 milliseconds in the optimized system, indicating a drastic improvement in query efficiency. Additionally, Redis helped stabilize response times for recurring queries in high-concurrency scenarios.

The technical validation demonstrated that the designed architecture not only meets scalability and performance requirements but also lays the foundation for the sustainable evolution of the monolithic system, enabled by the intrinsic modularity of microservices. The research provided valuable insights into how agile methodologies and the strategic implementation of microservices can transform legacy enterprise architectures, fostering more flexible, efficient, and adaptive technological environments.

Keywords:
Microservices, Distributed Indexing, Architectural Transformation, Legacy System Modernization, Real-Time Search, High Concurrency.

1. Introducción

La transformación digital ha impulsado a las organizaciones a replantear sus infraestructuras tecnológicas para responder de manera ágil a las demandas de un entorno empresarial cada vez más dinámico. Bajo este panorama, las arquitecturas tradicionales, especialmente las de tipo monolítico, han comenzado a mostrar signos de obsolescencia frente a los requerimientos de escalabilidad, tolerancia a fallos y velocidad de respuesta. Particularmente, en sistemas empresariales de reservas, donde la precisión y la inmediatez en la consulta de disponibilidad son factores críticos, las limitaciones estructurales del modelo monolítico pueden comprometer tanto la experiencia del usuario como la eficiencia operativa.

Frente a este panorama, las arquitecturas basadas en microservicios han emergido como una alternativa sólida, al permitir una gestión modular de las funcionalidades, el despliegue independiente de servicios y una mejor adaptabilidad frente a cargas variables. Este enfoque, complementado con técnicas de indexación distribuida y mecanismos de sincronización asincrónica, posibilita una respuesta más eficiente en escenarios de alta concurrencia y tráfico intensivo.

El presente trabajo se enmarca en este proceso de modernización arquitectónica, proponiendo una solución que convive con el sistema monolítico existente y, a la vez, introduce componentes optimizados para mejorar la consulta de disponibilidad en tiempo real. La propuesta se valida a través de un prototipo experimental que incorpora tecnologías como Elasticsearch, Redis y RabbitMQ, evaluando su impacto en términos de rendimiento, latencia y escalabilidad. La investigación se orienta así a ofrecer una base técnica para la transición progresiva hacia entornos más resilientes, modulares y eficientes, alineados con los desafíos actuales del sector tecnológico.

2. Determinación del Problema

Los sistemas de reservas empresariales que operan bajo arquitecturas monolíticas presentan limitaciones significativas en la gestión eficiente de consultas en tiempo real. Estas limitaciones se evidencian en latencias elevadas, dificultades para manejar accesos concurrentes y restricciones en la escalabilidad, afectando la disponibilidad y precisión de la información procesada.

La centralización de procesos en una única base de datos relacional genera cuellos de botella, especialmente en escenarios con alta demanda de consultas y actualizaciones constantes. Esta situación impacta la experiencia del usuario y la operatividad del sistema, dificultando la adaptación a los requerimientos actuales de disponibilidad inmediata y procesamiento eficiente de grandes volúmenes de datos.

3. Marco teórico referencial

A continuación, se exponen los fundamentos conceptuales y tecnológicos que sustentan el diseño de una arquitectura de microservicios con capacidades de indexación dinámica y distribuida, orientada a optimizar la búsqueda en tiempo real en sistemas de reservas empresariales. Se examinan los principios que rigen las arquitecturas monolíticas y de microservicios, resaltando sus diferencias, beneficios y desafíos en el contexto de la transformación digital.

Además, se analizan las estrategias de interoperabilidad, optimización de consultas y escalabilidad, con énfasis en herramientas especializadas como Elasticsearch, Redis y RabbitMQ. Finalmente, se presentan casos de estudio relevantes que ilustran la aplicación de estas tecnologías en la industria, estableciendo un marco de referencia sólido para la implementación de la solución propuesta.

3.1 Arquitectura de Microservicios y su Impacto en Sistemas de Reservas

Ilustración 1: Impacto de los microservicios en los sistemas de reserva (Mishra, Jaiswal, Prakash, & Barwal, 2022), (Horváth, Sakhnenko, & Gurbál, 2024)

La adopción de arquitecturas de microservicios ha transformado los sistemas empresariales modernos, permitiendo una mayor escalabilidad, flexibilidad y eficiencia operativa. En el contexto de los sistemas de reservas, esta arquitectura se presenta como una solución efectiva ante los desafíos de alta concurrencia y necesidad de respuesta en tiempo real. Como se aprecia en la Ilustración 1, los microservicios impactan directamente en estos sistemas a través de aspectos clave como su definición, evolución, comparación con arquitecturas monolíticas, patrones de integración y los desafíos asociados a la migración. La representación visual se elaboró a partir de los conceptos desarrollados por (Mishra, Jaiswal, Prakash, & Barwal, 2022), así como por (Horváth, Sakhnenko, & Gurbál, 2024), quienes analizan el papel de los microservicios en la modernización de infraestructuras críticas.

3.1.1 Definición y evolución de la arquitectura de microservicios

La arquitectura de microservicios constituye un enfoque de diseño de software basado en la descomposición de sistemas en servicios independientes, modulares y autónomos, que interactúan mediante interfaces bien definidas. A diferencia de la arquitectura monolítica, en la que todos los componentes del sistema se encuentran integrados en una única estructura, los microservicios permiten una mayor flexibilidad, escalabilidad y facilidad de mantenimiento, como lo demuestra su implementación en diversos sectores (Horváth, Sakhnenko, & Gurbál, 2024).

El concepto de microservicios emergió como una respuesta a las limitaciones de los sistemas monolíticos tradicionales, los cuales presentaban dificultades en términos de escalabilidad horizontal, integración continua y despliegue independiente. Investigaciones recientes han demostrado que la adopción de esta arquitectura ha sido impulsada por la necesidad de desarrollar aplicaciones más ágiles y resilientes, particularmente en entornos con altos volúmenes de datos y concurrencia, como los sistemas empresariales (Mishra, Jaiswal, Prakash, & Barwal, 2022).

La evolución de la arquitectura de microservicios se ha visto influenciada por múltiples avances tecnológicos. En sus primeras implementaciones, las aplicaciones

distribuían la lógica de negocio en módulos internos, pero seguían dependiendo de una infraestructura monolítica. Con la proliferación de contenedores y orquestadores como Docker y Kubernetes, se consolidó un modelo en el que cada microservicio puede ejecutarse de manera aislada, garantizando independencia en su escalabilidad y gestión, como se observa en implementaciones exitosas en sistemas de reservas y comercio electrónico (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022).

En la actualidad, la arquitectura de microservicios se ha convertido en un estándar para el desarrollo de aplicaciones escalables y de alta disponibilidad. Empresas tecnológicas líderes han implementado este enfoque para modernizar sus infraestructuras, aprovechando su capacidad para gestionar cargas de trabajo dinámicas y ofrecer una experiencia optimizada a los usuarios. Su adopción en sistemas empresariales representa una ventaja estratégica, al permitir búsquedas en tiempo real con un alto grado de precisión y eficiencia.

3.1.2 Comparación con arquitecturas monolíticas en sistemas empresariales

La arquitectura monolítica ha sido durante décadas el estándar en el desarrollo de sistemas empresariales debido a su simplicidad en el despliegue, mantenimiento y gestión de recursos. Este enfoque ha demostrado ser efectivo en entornos donde las aplicaciones requieren una alta cohesión interna y un ciclo de vida predecible, aunque presenta limitaciones en escalabilidad y mantenimiento (Horváth, Sakhnenko, & Gurbál, 2024). Con el auge de infraestructuras en la nube y la necesidad creciente de flexibilidad, modularidad y capacidad de evolución, las arquitecturas monolíticas han comenzado a evidenciar sus limitaciones. En particular, se ha identificado que dichas arquitecturas dificultan la agilidad en el diseño, desarrollo y despliegue de nuevos servicios, ya que cualquier cambio suele implicar modificaciones en todo el sistema. De igual manera, la rigidez de su estructura impide escalar componentes de forma independiente, lo que se traduce en un uso ineficiente de recursos y una compleja gestión operativa en entornos distribuidos (Di Francesco, Lago, & Malavolta, 2019).

En contraste, la arquitectura de microservicios propone una fragmentación de los sistemas en componentes independientes, cada uno con su propio entorno de ejecución y lógica de negocio. Esta separación permite que los servicios evolucionen de manera autónoma, facilitando su mantenimiento y mejorando la tolerancia a fallos. Adicionalmente, los microservicios pueden ser desplegados y escalados de forma individual, optimizando el uso de recursos en función de la demanda específica de cada módulo del sistema (Newman, 2021).

Característica	Arquitectura Monolítica	Arquitectura de Microservicios
Simplicidad de desarrollo	Simplicidad, dado que todo el código se encuentra en una única base de código.	Mayor complejidad debido a la división en múltiples servicios independientes.
Despliegue	Se despliega como una única unidad, facilitando la implementación	Cada servicio se despliega de manera independiente, permitiendo actualizar sin afectar al sistema completo.
Escalabilidad	Escalamiento vertical, lo que implica aumentar los recursos de toda la aplicación.	Escalamiento horizontal, permitiendo aumentar los recursos solo en los servicios que lo requieren.
Mantenimiento	Dificultad para actualizar módulos sin afectar toda la aplicación.	Tolerancia a fallos mejorada, ya que si un servicio falla, el resto del sistema puede seguir funcionando.
Tiempo de desarrollo	Más rápido para aplicaciones pequeñas y medianas debido a su menor complejidad inicial.	Mayor tiempo de desarrollo inicial debido a la necesidad de definir interfaces y mecanismos de comunicación entre servicios.
Comunicación interna	Comunicación interna rápida, ya que todos los módulos comparten memoria.	Requiere mecanismos eficientes de comunicación como REST, gRPC o mensajería (Kafka, RabbitMQ).
Ciclo de vida de desarrollo	Más predecible, con una única base de código y dependencias controladas.	Menos predecible, debido a la independencia de cada servicio y la necesidad de coordinación.
Flexibilidad y adaptabilidad	Más rígido; cualquier cambio estructural requiere modificaciones en toda la aplicación.	Alta flexibilidad, ya que cada servicio puede evolucionar de manera independiente.
Tolerancia a fallos	Un fallo puede afectar a todo el sistema.	La arquitectura distribuida permite mitigar fallos en servicios individuales.
Uso de recursos	Puede generar desperdicio de recursos si todos los módulos deben escalarse a la vez.	Optimización de recursos al escalar solo los servicios necesarios.
Optimización del rendimiento	Puede ser eficiente en cargas de trabajo estables y predecibles.	Se adapta mejor a cargas de trabajo variables y demanda fluctuante.

Tabla 1: Comparación entre Arquitectura Monolítica y de Microservicios.

Sin embargo, el uso de microservicios introduce nuevos desafíos, como la necesidad de mecanismos eficientes para la comunicación entre servicios y la complejidad en la orquestación del sistema. A pesar de estos retos, análisis recientes han demostrado que la adopción de microservicios en sistemas empresariales ha permitido mejorar la flexibilidad y la capacidad de adaptación ante cambios en los requerimientos del negocio, optimizando además el rendimiento en cargas de trabajo variables (Horváth, Sakhnenko, & Gurbál, 2024), (Mishra, Jaiswal, Prakash, & Barwal, 2022).

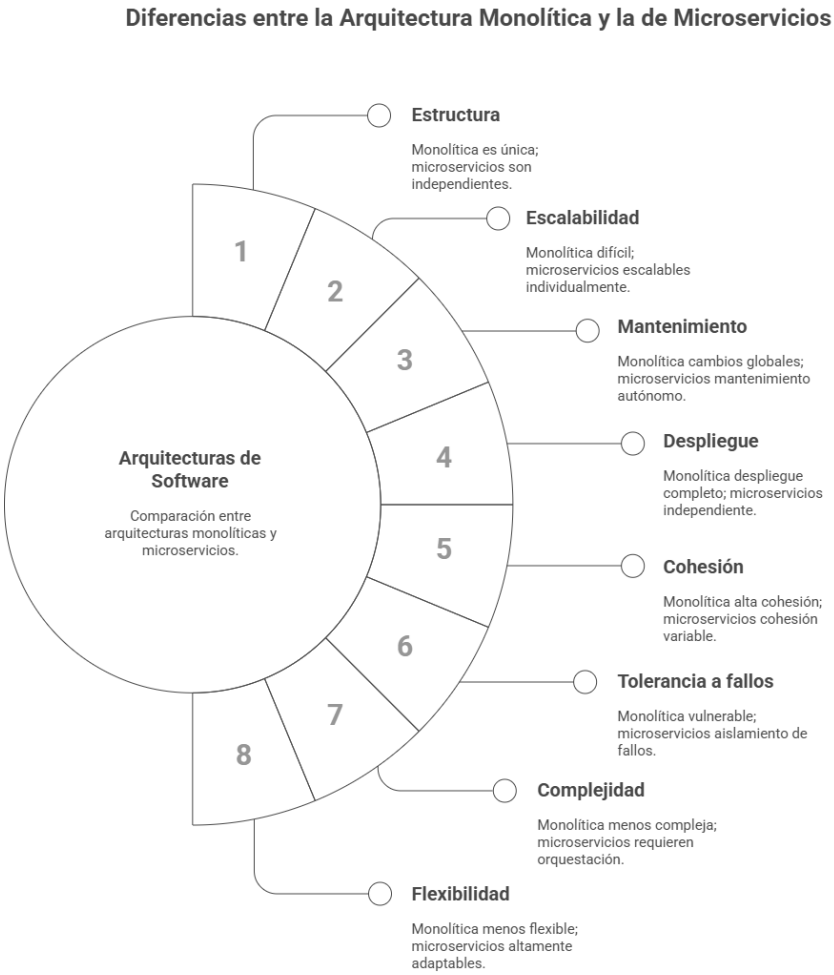

Ilustración 2: Diferencias entre la Arquitectura Monolítica y la de Microservicios (Horváth, Sakhnenko, & Gurbál, 2024)

En entornos empresariales, las arquitecturas monolíticas han sido tradicionalmente preferidas por su simplicidad y facilidad de despliegue, especialmente en aplicaciones pequeñas y medianas. Sin embargo, frente a la creciente necesidad de escalabilidad, resiliencia y flexibilidad, los microservicios emergen como una

alternativa más robusta. Como se muestra en la Ilustración 2, los microservicios permiten el escalamiento individual de componentes, tolerancia a fallos mediante aislamiento de servicios y una mayor adaptabilidad ante cambios, lo que representa ventajas significativas frente a la rigidez y el acoplamiento fuerte del enfoque monolítico (Horváth, Sakhnenko, & Gurbál, 2024).

3.1.3 Patrones de Migración e Integración de microservicios con sistemas monolíticos

La coexistencia entre arquitecturas monolíticas y microservicios representa un desafío recurrente en los procesos de modernización de sistemas empresariales. En la mayoría de los casos, los sistemas heredados no pueden ser reemplazados de manera inmediata, lo que obliga a adoptar estrategias de transición progresiva que aseguren la continuidad operativa mientras se avanza hacia una arquitectura más flexible y escalable. Entre los patrones más utilizados para facilitar esta transición destaca el Strangler Pattern, el cual propone la migración gradual de componentes de un sistema monolítico a microservicios. En este enfoque, los nuevos servicios se desarrollan de forma independiente y se integran con el sistema existente mediante una capa intermedia, como un API Gateway, que redirige las solicitudes según la disponibilidad de los servicios en la nueva arquitectura (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022).

Ilustración 3: Pasos del Patrón Strangler
(Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022)

Por otro lado, un patrón ampliamente aplicado es el de Arquitectura Orientada a Eventos (Event-Driven Architecture, EDA). En este modelo, los microservicios se comunican mediante eventos asincrónicos gestionados a través de colas de mensajes, como RabbitMQ o Kafka, en lugar de realizar llamadas directas entre servicios. Este enfoque reduce significativamente las dependencias entre componentes, ya que los productores de eventos no necesitan conocer a los consumidores. La introducción de un intermediario de mensajes permite desacoplar los flujos de datos, fortaleciendo la resiliencia del sistema, mejorando la escalabilidad y permitiendo la actualización independiente de los servicios. Múltiples implementaciones recientes en entornos empresariales han demostrado su eficacia para integrar sistemas monolíticos con microservicios, especialmente en escenarios de sincronización de datos (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022), (Mishra, Jaiswal, Prakash, & Barwal, 2022).

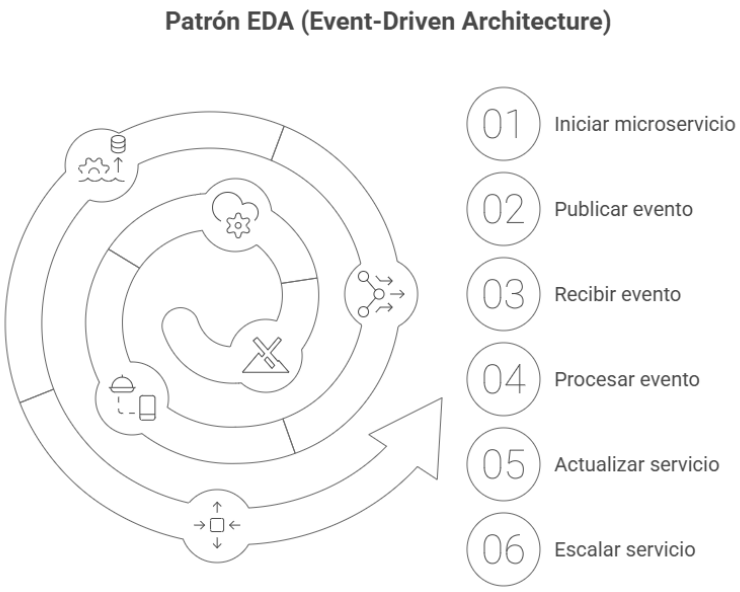

Ilustración 4: Patrón EDA (Event-Driven Architecture)
(Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022), (Mishra, Jaiswal, Prakash, & Barwal, 2022)

Adicionalmente, la utilización de Adapters y Capas Anti-Corrupción (Anti-Corruption Layers) es una práctica recomendada cuando se requiere integrar microservicios con sistemas monolíticos que poseen modelos de datos incompatibles. En el enfoque propuesto por (Li, Ma, & Lu, 2020), estos componentes intermedios actúan

como traductores que permiten la interoperabilidad entre subsistemas sin necesidad de alterar la estructura interna del sistema heredado. Como se representa en la Ilustración 5, estos componentes intermedios facilitan la conversión y traducción de datos entre ambos entornos, asegurando la interoperabilidad sin necesidad de modificar la estructura interna del sistema heredado.

Ilustración 5: Adaptadores y Capas de Anticorrupción
(Li, Ma, & Lu, 2020)

La implementación adecuada de estos patrones permite a las organizaciones modernizar sus infraestructuras tecnológicas de forma progresiva, minimizando riesgos técnicos y operativos. Sin embargo, su éxito depende de un análisis detallado de las dependencias existentes, una gestión cuidadosa del tráfico entre componentes y una planificación estratégica que considere tanto los aspectos técnicos como organizacionales del proceso de migración.

3.1.4 Desafíos de la migración de arquitecturas monolíticas a microservicios

El proceso de migración de una arquitectura monolítica a un modelo basado en microservicios implica una serie de desafíos tanto técnicos como organizacionales que deben ser gestionados adecuadamente para garantizar el éxito de la transición.

Uno de los desafíos más relevantes en la adopción del estilo arquitectónico basado en microservicios es la correcta descomposición del sistema en servicios independientes. Según se evidencia en la literatura, esta tarea requiere un análisis exhaustivo de las capacidades del negocio y de las dependencias existentes entre componentes. Una partición inadecuada puede provocar problemas de granularidad y diseño, afectando la mantenibilidad, cohesión e incluso la escalabilidad del sistema resultante (Di Francesco, Lago, & Malavolta, 2019).

Otro desafío relevante en el proceso de migración de arquitecturas monolíticas a microservicios es la gestión de datos distribuidos. Mientras que en un sistema monolítico todas las operaciones suelen depender de una base de datos centralizada, en un entorno basado en microservicios, cada componente administra su propio almacenamiento de forma independiente. Esta independencia introduce dificultades en términos de consistencia, sincronización y transacciones distribuidas.

Ilustración 6:Patrones de Persistencia y Consulta Event Sourcing y CQRS (Mishra, Jaiswal, Prakash, & Barwal, 2022)

En este contexto, y como complemento a los patrones de integración vistos anteriormente, se aplican estrategias específicas como Event Sourcing, que consiste en almacenar los cambios del sistema como una secuencia de eventos en lugar de guardar directamente el estado final, y Command Query Responsibility Segregation (CQRS), que separa los modelos de lectura y escritura para optimizar el rendimiento y la escalabilidad. Su implementación resulta fundamental en escenarios de alto tráfico, como lo demuestran diversos estudios recientes (Mishra, Jaiswal, Prakash, & Barwal, 2022), (Horváth, Sakhnenko, & Gurbál, 2024).

Asimismo, la migración hacia microservicios incrementa la complejidad operativa, al requerir herramientas especializadas para el despliegue, monitoreo y tolerancia a fallos. Tecnologías como Docker, Kubernetes e Istio han sido ampliamente adoptadas para facilitar la orquestación y gestión automatizada de servicios, tal como se evidencia en diversos estudios de migración arquitectónica (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022), (Horváth, Sakhnenko, & Gurbál, 2024).

La migración a microservicios trasciende el ámbito técnico para exigir una reconfiguración organizacional, a menudo alineada con la Ley de Conway. A diferencia del enfoque monolítico con equipos estructurados por capas tecnológicas (front-end, back-end, bases de datos), la arquitectura de microservicios promueve la creación de equipos multidisciplinarios y verticales, cada uno con autonomía y responsabilidad de extremo a extremo (end-to-end) sobre un servicio de negocio específico (Newman, 2021).A pesar de los desafíos que implica la migración de sistemas legados, la revisión sistemática realizada por (Hasan, Osman, Admodisastro, & Muhammad, 2023) subraya que la adopción de arquitecturas de microservicios conlleva beneficios significativos, en particular en términos de escalabilidad, resiliencia y capacidad de innovación. Para mitigar los riesgos asociados a este proceso, los autores recomiendan adoptar un enfoque iterativo de modernización, priorizando los componentes más críticos del sistema y aplicando pruebas progresivas que permitan validar la estabilidad y calidad de la nueva arquitectura.

3.2 Anatomía de la Arquitectura Monolítica Empresarial

Tras haber contrastado las arquitecturas monolíticas con los microservicios, esta sección profundiza en los patrones de diseño y componentes tecnológicos que caracterizan a los sistemas empresariales tradicionales. La arquitectura monolítica ha sido el enfoque predominante en el desarrollo de sistemas empresariales debido a su simplicidad estructural y facilidad de implementación. Comprender esta anatomía interna es fundamental para diagnosticar las causas raíz de sus limitaciones y justificar el enfoque de modernización propuesto en este trabajo.

3.2.1 Componentes clave de los sistemas monolíticos

Las arquitecturas monolíticas están conformadas por distintos componentes que interactúan dentro de una única aplicación. Entre los más relevantes se encuentran los patrones de diseño y las tecnologías que han sido ampliamente utilizadas en este tipo de sistemas.

Uno de los modelos más adoptados en la estructuración de aplicaciones monolíticas es el patrón Modelo-Vista-Controlador (MVC), como se observa en la Ilustración 7, el cual permite separar la lógica de presentación de la lógica de negocio y el acceso a datos. Aunque este enfoque sigue siendo utilizado en ciertos sistemas empresariales, publicaciones han demostrado que su integración con arquitecturas modernas presenta desafíos en términos de modularidad y escalabilidad (Mishra, Jaiswal, Prakash, & Barwal, 2022). Esta segmentación facilita la gestión del código y mejora la reutilización de componentes dentro de la misma aplicación.

Ilustración 7: Componentes del Patrón MVC escalabilidad
(Mishra, Jaiswal, Prakash, & Barwal, 2022)

En entornos empresariales, es común el uso de Java Enterprise Edition (JEE) como plataforma para el desarrollo de aplicaciones monolíticas. JEE proporciona un conjunto de especificaciones que permiten la construcción de sistemas robustos y escalables. No obstante, su integración con entornos de microservicios requiere adaptaciones significativas, ya que los enfoques monolíticos pueden generar dependencias difíciles de desacoplar (Mishra, Jaiswal, Prakash, & Barwal, 2022). Un componente más utilizado es el Enterprise Java Bean (EJB), el cual ofrece una solución para la gestión de transacciones y la implementación de lógica de negocio en entornos distribuidos.

Ilustración 8: Estructura de EJB en JEE
(Mishra, Jaiswal, Prakash, & Barwal, 2022)

Por otro lado, el manejo de datos en arquitecturas monolíticas suele estar basado en frameworks como Java Persistence API (JPA), que facilita la persistencia de información mediante el uso de entidades y mapeo objeto-relacional. JPA permite la integración con bases de datos relacionales, proporcionando una interfaz estándar para la manipulación de datos dentro de la aplicación.

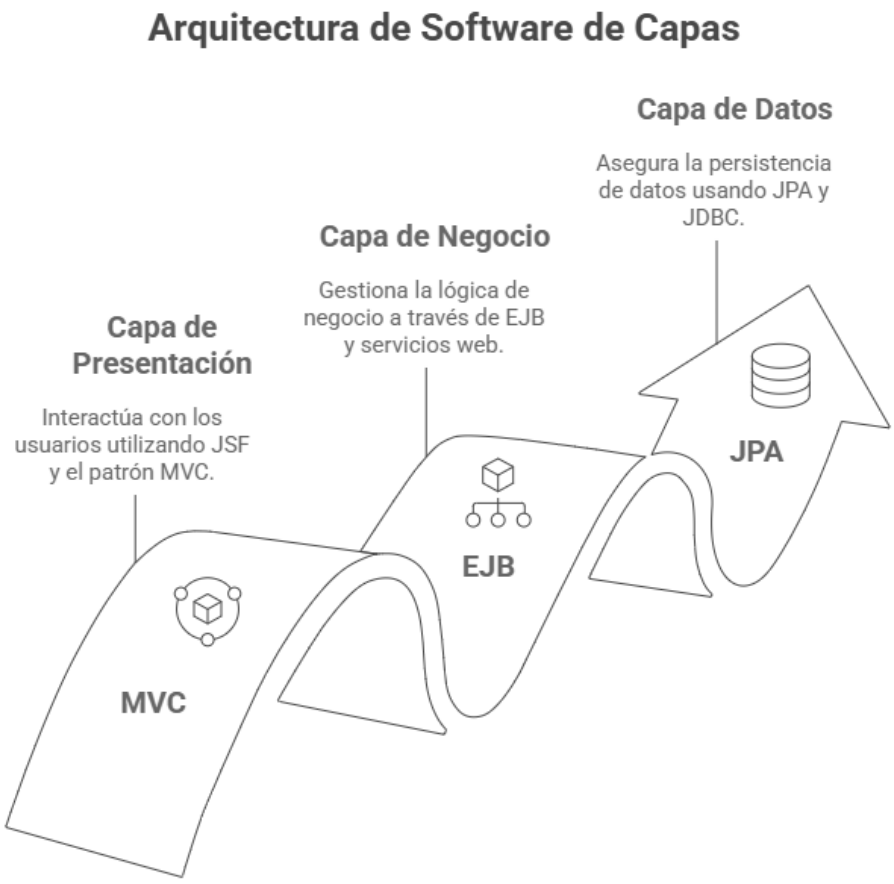

Ilustración 9: Arquitectura de Software de Capas

El uso de estos componentes ha permitido la construcción de sistemas monolíticos eficientes y estructurados. No obstante, el crecimiento de las aplicaciones y la demanda por mayor escalabilidad han evidenciado ciertas limitaciones que han impulsado la exploración de nuevas arquitecturas.

3.2.2 Problemas de mantenimiento y flexibilidad en sistemas monolíticos

Uno de los aspectos críticos de las arquitecturas monolíticas es la dificultad en su mantenimiento y evolución a lo largo del tiempo. Esta rigidez estructural ha sido ampliamente documentada en investigaciones sobre la transición de sistemas heredados a modelos de microservicios (Mishra, Jaiswal, Prakash, & Barwal, 2022). Como todos los módulos están integrados en una sola aplicación, cualquier cambio en el sistema requiere la modificación del código base completo. Esto incrementa el riesgo de generar errores en funcionalidades no relacionadas con la actualización implementada.

El ciclo de desarrollo en sistemas monolíticos suele ser más largo y complejo, ya que cada nueva versión de la aplicación debe ser probada en su totalidad antes de ser desplegada. Esta dependencia entre módulos hace que los equipos de desarrollo deban coordinar sus actividades con mayor rigurosidad, lo que ralentiza la entrega de nuevas funcionalidades y dificulta la adopción de metodologías ágiles.

Desde el punto de vista de la flexibilidad, los sistemas monolíticos presentan limitaciones para la incorporación de nuevas tecnologías. Comparaciones entre modelos monolíticos y distribuidos han revelado que las arquitecturas tradicionales requieren actualizaciones completas, mientras que los microservicios permiten cambios incrementales sin afectar la operatividad global (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). La necesidad de mantener compatibilidad con las versiones anteriores del software impide una actualización progresiva de los componentes, lo que puede derivar en la obsolescencia de la plataforma en un corto periodo de tiempo.

Asimismo, la gestión de fallos en una arquitectura monolítica puede representar un problema significativo. Dado que todos los módulos están interconectados, un error en una parte del sistema puede afectar el funcionamiento global de la aplicación. Estudios en entornos empresariales han puesto en evidencia que la resiliencia de los microservicios reduce significativamente el impacto de errores en la disponibilidad del sistema (Horváth, Sakhnenko, & Gurbál, 2024).

3.3 Indexación Dinámica y Distribuida en la Optimización de Búsquedas

La indexación dinámica y distribuida se ha convertido en un componente esencial en los sistemas de información modernos, permitiendo mejorar el rendimiento de consultas y optimizar la recuperación de datos en tiempo real. Su valor en el contexto empresarial es particularmente evidente, al garantizar respuestas eficientes en entornos de alta concurrencia (He, 2020), (Wang, Yu, & Tang, 2020). Dicha capacidad se vuelve crítica en los sistemas de reservas, un sector caracterizado por una demanda constante y volátil. En este escenario, gestionar eficientemente grandes volúmenes de datos impacta de forma directa en la experiencia del usuario y en la ventaja competitiva del negocio.

3.3.1 Importancia de la indexación en la optimización de búsquedas

La búsqueda eficiente de información en grandes volúmenes de datos ha sido históricamente un desafío en los sistemas empresariales, especialmente en aquellos donde la actualización de datos es constante y las consultas requieren tiempos de respuesta mínimos (He, 2020). La indexación juega un papel crucial en la optimización de consultas, permitiendo organizar los datos de manera estructurada para facilitar su recuperación de manera rápida y eficiente.

La indexación tradicional basada en bases de datos relacionales resulta insuficiente en escenarios donde los datos cambian dinámicamente y requieren una actualización constante. En este sentido, los motores de búsqueda modernos han incorporado técnicas de indexación en estructuras distribuidas, reduciendo la latencia y mejorando la escalabilidad del sistema. La utilización de índices distribuidos permite fragmentar la información en múltiples nodos, reduciendo la carga de procesamiento y optimizando la ejecución de consultas concurrentes en sistemas de gran escala. En particular, trabajos académicos sobre búsqueda personalizada han demostrado que la combinación de indexación distribuida y

algoritmos de relevancia mejora significativamente la precisión y eficiencia de recuperación de información (He, 2020), (Mishra, Jaiswal, Prakash, & Barwal, 2022).

3.3.2 Tecnologías y herramientas para indexación distribuida

En la actualidad, diversas herramientas especializadas permiten implementar esquemas de indexación distribuida. Entre las más utilizadas se encuentran Elasticsearch, Opensearch, Amazon CloudSearch y Apache Solr, cada una con características que las hacen adecuadas para diferentes tipos de sistemas y volúmenes de datos.

Característica	Elasticsearch	Opensearch	CloudSearch	Apache Solr
Modelo de despliegue	Distribuido, auto-gestionado o en la nube (ElasticCloud, AWS, Azure)	Distribuido, auto-gestionado o en AWS	Solución gestionada en la nube (AWS)	Distribuido, auto-gestionado
Motor base	Basado en Apache Lucene	Basado en Apache Lucene (Bifurcación de Elasticsearch)	AWS CloudSearch Engine	Basado en Apache Lucene
Escalabilidad	Alta escalabilidad mediante clústeres distribuidos	Similar a Elasticsearch, con mejoras en gestión y tolerancia a fallos	Escalabilidad automática gestionada por AWS	Alta escalabilidad mediante shards y réplicas
Latencia y rendimiento	Optimizado para búsquedas en tiempo real y alta concurrencia	Optimizado con enfoque en estabilidad y menor latencia	Rendimiento variable según la demanda, gestionado por AWS	Alto rendimiento, pero requiere más configuración
Seguridad	Funcionalidades básicas (requiere plugins para seguridad avanzada)	Mejoras en seguridad integrada (autenticación, control de acceso)	Seguridad gestionada por AWS	Seguridad configurable, pero depende del usuario
Optimización de costos	Requiere gestión de infraestructura, costos variables	Mejor optimización de costos en entornos empresariales	Basado en pago por uso (AWS), sin gestión de infraestructura	Costos variables, pero más flexible en hardware propio

Tolerancia a fallos	Implementa replicación y shards para recuperación automática	Mejor manejo de tolerancia a fallos en clústeres	Alta disponibilidad gestionada por AWS	Soporta replicación y recuperación manual
Personalización y extensibilidad	Amplio soporte de plugins y personalización	Compatible con la mayoría de plugins de Elasticsearch	Limitado a configuración estándar en AWS	Alta capacidad de personalización mediante configuraciones avanzadas
Casos de uso ideales	Aplicaciones de búsqueda en tiempo real, análisis de logs, big data	Entornos empresariales con necesidad de seguridad y estabilidad	Empresas que buscan una solución sin administración de infraestructura	Aplicaciones empresariales con búsqueda avanzada y personalizable

Tabla 2: Comparación entre Elasticsearch, OpenSearch, CloudSearch y Apache Solr

Elasticsearch se selecciona como motor de búsqueda debido a su capacidad de indexación dinámica y escalabilidad en clústeres distribuidos, optimizando la búsqueda en tiempo real de disponibilidad en sistemas empresariales. Su alto rendimiento en consultas concurrentes y flexibilidad en personalización lo convierten en la mejor opción para coexistir con una arquitectura monolítica sin comprometer eficiencia ni estabilidad.

Ilustración 10: Selección de motor de búsqueda basado en casos de uso

3.3.3 Algoritmos de búsqueda y recuperación de información en bases de datos distribuidas

Los algoritmos de búsqueda juegan un papel fundamental en la eficiencia de los sistemas de indexación distribuida. Diversas investigaciones han demostrado que la elección del algoritmo adecuado depende en gran medida del tipo de datos y del volumen de consultas concurrentes en el sistema (He, 2020), (Mishra, Jaiswal, Prakash, & Barwal, 2022). Entre los más utilizados se encuentran los árboles B+, los índices invertidos y los algoritmos de hashing distribuido, cada uno diseñado para optimizar el acceso a los datos y reducir el tiempo de consulta.

El índice invertido ha demostrado ser una de las técnicas más efectivas, al permitir la recuperación rápida de documentos que contienen términos específicos sin necesidad de recorrer grandes volúmenes de datos. Esta técnica, pilar de motores como Elasticsearch y Solr, mejora significativamente la velocidad de búsqueda y la capacidad de respuesta del sistema. Investigaciones sobre personalización de búsqueda han revelado, además, que la combinación de índices invertidos con algoritmos de relevancia incrementa la precisión de los resultados en sistemas de recomendación (He, 2020), (Horváth, Sakhnenko, & Gurbál, 2024).

Para que estos índices operen a gran escala, es indispensable aplicar técnicas de distribución como el sharding (fragmentación) y la replicación. El sharding permite dividir un índice masivo en múltiples nodos para paralelizar las consultas, mientras que la replicación crea copias de seguridad para garantizar la alta disponibilidad y tolerancia a fallos. La implementación de estas estrategias en motores como Elasticsearch y Apache Solr ha sido clave para lograr una escalabilidad eficiente en plataformas de comercio electrónico y sistemas de reservas (Mishra et al., 2022), (Rahmatulloh et al., 2022).

3.4 Optimización de Búsquedas en Sistemas de Reservas

La optimización de consultas en sistemas empresariales modernos representa un componente estratégico para mejorar el rendimiento general de la plataforma. En entornos donde la disponibilidad de recursos cambia constantemente y los volúmenes de datos son elevados, garantizar respuestas rápidas y precisas se convierte en un factor diferenciador. Esta sección analiza los elementos que influyen en la eficiencia de las búsquedas, así como las técnicas empleadas para reducir la latencia y maximizar la experiencia del usuario.

3.4.1 Impacto del rendimiento y latencia en la experiencia del usuario

En la arquitectura de un sistema de reservas, la latencia no debe considerarse solo un parámetro técnico, sino el factor que define directamente la calidad de la experiencia del usuario y la viabilidad comercial del servicio. Tiempos de respuesta elevados generan demoras en la visualización de la disponibilidad, afectando la toma de decisiones del cliente y reduciendo la competitividad del negocio.

Investigaciones en el campo de la usabilidad web han demostrado que incluso una diferencia de milisegundos en los tiempos de respuesta puede influir de manera significativa en la tasa de conversión y en la fidelización de los clientes. Dado que los sistemas de reservas manejan grandes volúmenes de datos en tiempo real, la optimización mediante técnicas de indexación distribuida y almacenamiento en caché se vuelve indispensable. Estas estrategias permiten reducir drásticamente la latencia y garantizar un rendimiento estable incluso bajo condiciones de alta concurrencia, lo cual es esencial para la viabilidad del sistema (He, 2021), (Rahmatulloh et al., 2022).

3.4.2 Técnicas de optimización de consultas en sistemas empresariales

Para mejorar el rendimiento de las consultas en sistemas de reservas, se han desarrollado diversas técnicas de optimización basadas en estrategias avanzadas de indexación y modelos de procesamiento distribuidos (Mishra, Jaiswal, Prakash, & Barwal, 2022), (Horváth, Sakhnenko, & Gurbál, 2024). Entre las más destacadas se encuentran (como se ejemplifica en la Ilustración 11):

- Indexación avanzada: Implementación de estructuras optimizadas para facilitar la recuperación de información sin necesidad de recorrer grandes volúmenes de datos.
- Optimización de consultas SQL: Reestructuración de consultas y uso de índices en bases de datos relacionales para mejorar el rendimiento en operaciones de búsqueda y filtrado.
- Estrategias de particionamiento: División de datos en fragmentos más pequeños para distribuir la carga de procesamiento y mejorar la velocidad de acceso.

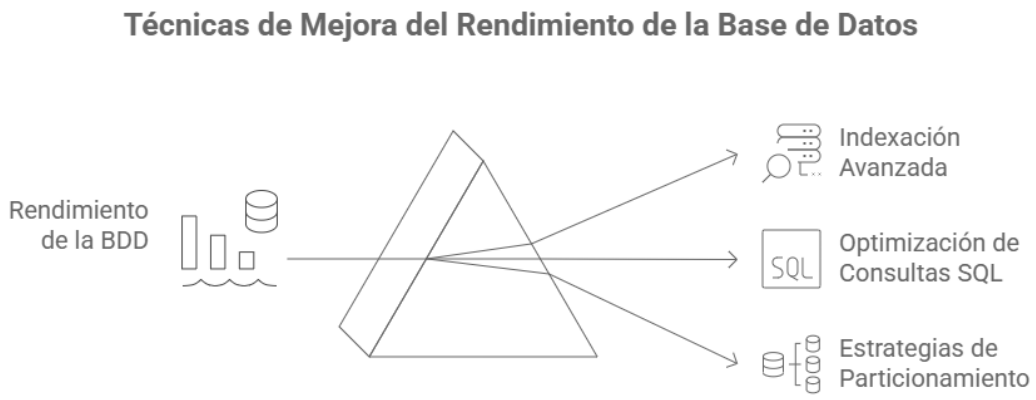

Ilustración 11: Técnicas de Mejora del Rendimiento de la Base de Datos (Mishra, Jaiswal, Prakash, & Barwal, 2022), (Horváth, Sakhnenko, & Gurbál, 2024)

Estas técnicas han sido adoptadas en múltiples sistemas empresariales con el objetivo de reducir los tiempos de respuesta y mejorar la eficiencia en la recuperación de datos.

3.4.3 Uso de caché distribuido para mejorar la velocidad de respuesta

El almacenamiento en caché es una estrategia clave para optimizar el rendimiento de sistemas que requieren respuestas en tiempo real. Esta técnica consiste en almacenar temporalmente los resultados de consultas frecuentes en una capa de memoria de alta velocidad, evitando así múltiples accesos a las fuentes de datos principales, que son más lentas. Su implementación en sistemas de reservas resulta fundamental para mejorar la escalabilidad y reducir la carga sobre las bases de datos centrales (He, 2020), (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022).

Aunque tradicionalmente se asocia el caché con datos estáticos, su aplicación es crucial en sistemas con información dinámica, como la disponibilidad en tiempo real, si bien el enfoque estratégico cambia. En estos escenarios, el objetivo no es almacenar datos a largo plazo, sino absorber picos de alta concurrencia y proteger los sistemas de backend. Esto se logra mediante técnicas como el uso de un Tiempo de Vida (TTL) muy corto, o la invalidación proactiva de la caché, donde un evento ordena la eliminación del dato obsoleto. Por tanto, la tecnología de caché seleccionada debe soportar estas operaciones de manera eficiente.

Existen diversas tecnologías diseñadas para el almacenamiento en caché dentro de arquitecturas distribuidas, cada una con características específicas orientadas a optimizar el acceso y procesamiento de datos en tiempo real:

- Redis: Reconocido por su capacidad de procesamiento en memoria y su compatibilidad con estructuras de datos avanzadas, lo que permite optimizar la búsqueda y almacenamiento de información temporal. Es ampliamente utilizado en aplicaciones web y sistemas de alta concurrencia debido a su capacidad para manejar millones de operaciones por segundo.
- Memcached: Popular en aplicaciones que requieren un almacenamiento en caché liviano y de baja latencia. Su estructura simple y eficiente lo convierte en una opción ideal para reducir la carga en bases de datos cuando las consultas son repetitivas y no requieren operaciones complejas.

- ElastiCache: Servicio administrado en la nube de Amazon Web Services (AWS), basado en Redis y Memcached, que permite escalar el almacenamiento en caché sin necesidad de administrar directamente la infraestructura subyacente.
- Apache Ignite y Infinispan: Tecnologías diseñadas para almacenamiento en caché en sistemas distribuidos, con soporte para computación en memoria y persistencia de datos, facilitando la escalabilidad en entornos empresariales.
- Ehcache: Solución ampliamente utilizada en aplicaciones basadas en Java, especialmente en entornos donde la optimización del acceso a bases de datos es un requisito clave.

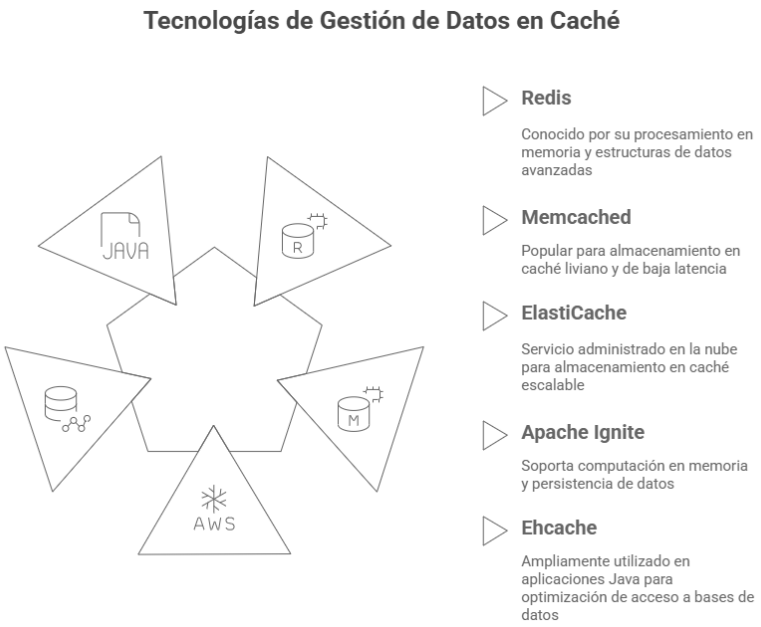

Ilustración 12: Tecnologías de Gestión de Datos en Caché

Trabajos académicos han demostrado que la optimización de la caché en sistemas de búsqueda mejora el rendimiento en hasta un 40 % en escenarios de alta concurrencia (Horváth, Sakhnenko, & Gurbál, 2024), (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). Un caso típico es el almacenamiento de los resultados de búsquedas de vuelos en un sistema de reservas: en lugar de calcular la disponibilidad en cada solicitud, el sistema puede recuperar los resultados desde la caché, lo que disminuye la carga en el servidor y reduce la latencia.

Además, el uso de estrategias avanzadas como caché con expiración automática y caché basada en políticas de actualización permite mantener la información siempre actualizada, garantizando que los usuarios reciban datos precisos sin comprometer la eficiencia del sistema. En sistemas de reservas donde la disponibilidad cambia constantemente, la configuración de tiempos de expiración cortos y la actualización proactiva de la caché son factores críticos para garantizar la coherencia de los datos almacenados en memoria. La implementación de caché distribuido en sistemas de reservas permite mejorar la velocidad de respuesta, optimizar la eficiencia del sistema y garantizar una mejor experiencia para el usuario final en entornos de alta concurrencia.

3.4.4 Casos de estudio en la implementación de búsqueda optimizada

La optimización de búsquedas en sistemas de reservas ha sido un factor determinante en el éxito de plataformas líderes en el sector turístico, hotelero y de transporte. La adopción de microservicios y técnicas de indexación distribuida ha permitido una mejor gestión de la disponibilidad de recursos en entornos de alta demanda (He, 2020), (Mishra, Jaiswal, Prakash, & Barwal, 2022). Diversas empresas han implementado estrategias avanzadas de indexación y almacenamiento en caché para mejorar la eficiencia en la recuperación de información y garantizar tiempos de respuesta óptimos en entornos de alta demanda.

Un ejemplo representativo es el caso de Expedia, una de las plataformas de reservas de viajes más utilizadas a nivel mundial. Para optimizar la búsqueda de vuelos y hoteles, la compañía ha adoptado una arquitectura basada en microservicios que utiliza Elasticsearch para indexar millones de registros y Redis para almacenar consultas frecuentes. Datos recopilados han demostrado que la combinación de indexación distribuida y almacenamiento en caché puede reducir la latencia en hasta un 70 % en plataformas de reservas masivas (He, 2020), (Horváth, Sakhnenko, & Gurbál, 2024). Esto ha permitido reducir drásticamente la latencia en la recuperación de información, proporcionando resultados en milisegundos incluso en momentos de alta concurrencia.

Otro caso emblemático es Booking.com, donde la optimización de búsquedas ha sido clave para mejorar la experiencia del usuario. La empresa ha implementado estrategias de preprocesamiento de consultas, utilizando caché distribuido para almacenar resultados de búsqueda basados en patrones de consulta recurrentes. Esta técnica ha permitido reducir la carga sobre la base de datos y mejorar el tiempo de respuesta sin comprometer la precisión de los datos mostrados a los usuarios.

En el sector del transporte, Uber ha implementado técnicas avanzadas de indexación y particionamiento de datos para optimizar la búsqueda y asignación de conductores en tiempo real. A través de un sistema basado en Apache Cassandra y Elasticsearch, la plataforma puede gestionar millones de solicitudes simultáneamente, garantizando una respuesta rápida y eficiente incluso en ciudades con alta demanda.

Por otro lado, compañías de venta de entradas y espectáculos han optimizado sus sistemas de reservas para evitar colapsos durante eventos de gran demanda. La integración de algoritmos de predicción y balanceo de carga ha sido clave en estos entornos de alta concurrencia (Mishra, Jaiswal, Prakash, & Barwal, 2022), (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). Un ejemplo de ello es Ticketmaster, que utiliza un enfoque híbrido de caché distribuido y replicación de datos en múltiples servidores para manejar picos de tráfico sin afectar la disponibilidad de los boletos. Gracias a esta estrategia, la empresa ha logrado gestionar lanzamientos de eventos con alto tráfico, asegurando tiempos de respuesta estables y evitando la saturación del sistema.

Estos casos de estudio demuestran que la combinación de técnicas de indexación distribuida, almacenamiento en caché y optimización de consultas es fundamental para garantizar la escalabilidad y el rendimiento en sistemas de reservas. La aplicación de estas estrategias permite a las empresas mejorar la eficiencia operativa y proporcionar a los usuarios experiencias de búsqueda rápidas y precisas, fortaleciendo así su competitividad en el mercado digital.

3.5 Interoperabilidad entre Microservicios y Sistemas Monolíticos

La evolución de las arquitecturas empresariales ha dado lugar a escenarios en los que conviven sistemas monolíticos y microservicios, una situación habitual en organizaciones con infraestructuras heredadas. Esta coexistencia ha sido ampliamente analizada en el contexto de la transición hacia entornos distribuidos, donde se han propuesto estrategias efectivas para garantizar la interoperabilidad y mitigar los riesgos de migración (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). Estas aproximaciones se ilustran mediante casos como los presentados en la Ilustración 13, que evidencian soluciones técnicas para entornos híbridos.

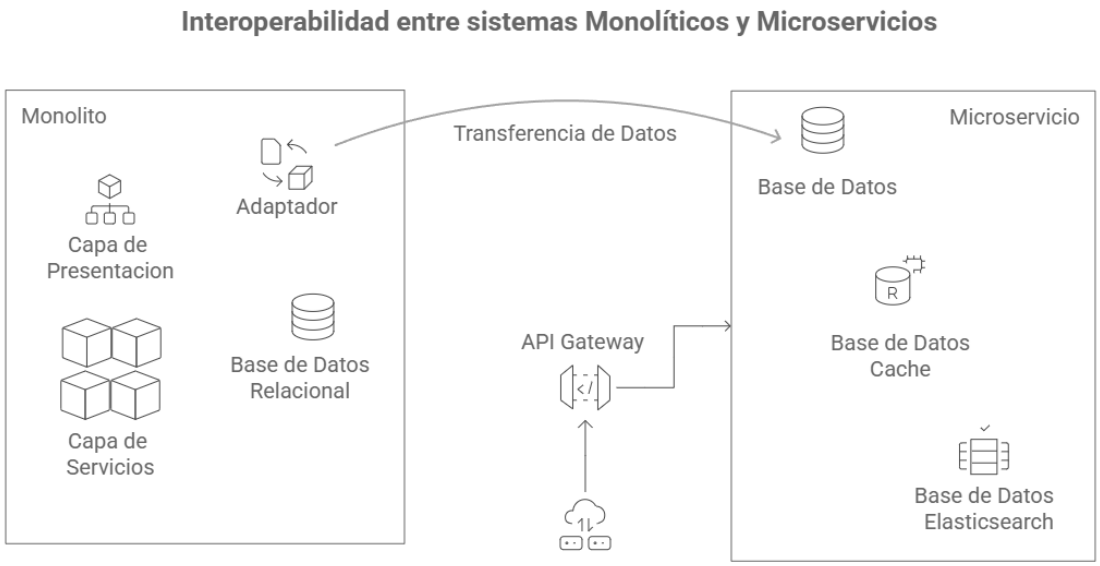

Ilustración 13: Interoperabilidad entre sistemas Monolíticos y Microservicios (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022)

Dado que una migración completa hacia microservicios puede ser compleja y costosa, es fundamental establecer estrategias de interoperabilidad que permitan integrar ambos enfoques sin comprometer la estabilidad del sistema. La capacidad de comunicar eficientemente microservicios con sistemas monolíticos garantiza una transición gradual, evitando interrupciones en la operación y maximizando la reutilización de recursos existentes.

3.5.1 Estrategias para la coexistencia de arquitecturas monolíticas y microservicios

La coexistencia entre sistemas monolíticos y microservicios requiere la implementación de estrategias que permitan la comunicación fluida y la sincronización de datos sin afectar la integridad del sistema. Diversos trabajos de investigación han demostrado que la elección de una estrategia adecuada puede determinar el éxito de la modernización de infraestructuras empresariales (Horváth, Sakhnenko, & Gurbál, 2024). Entre las metodologías más empleadas destacan:

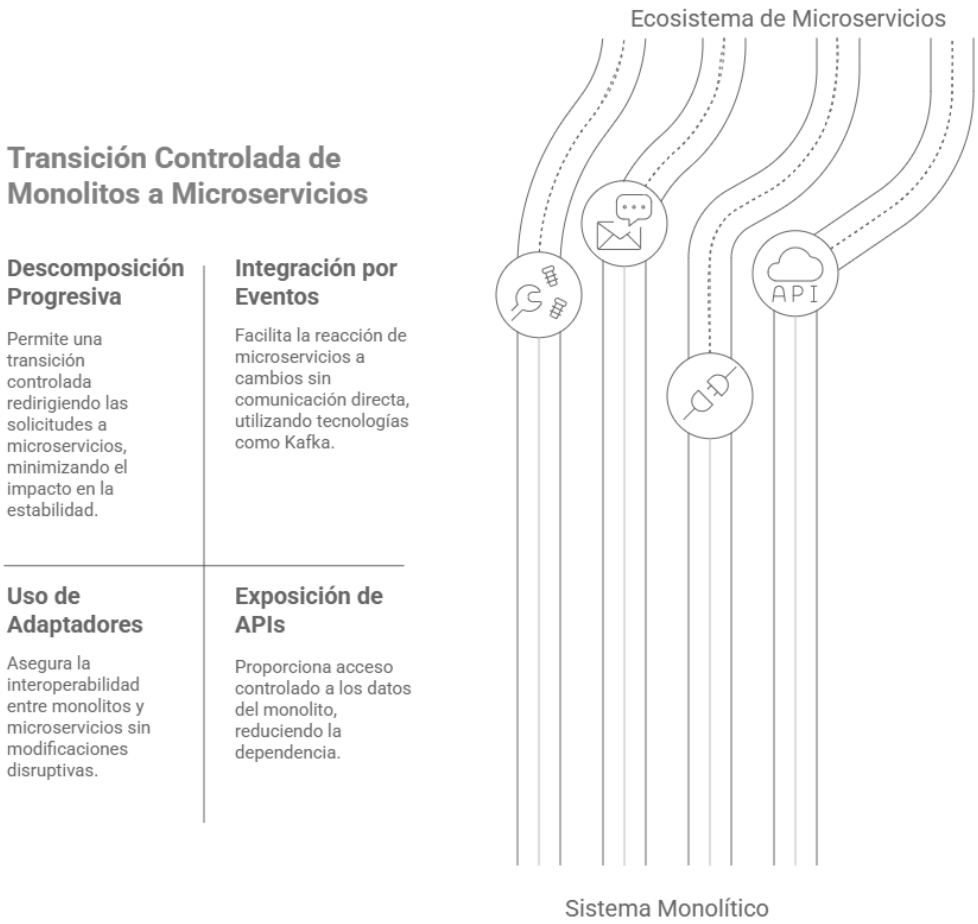

Ilustración 14: Transición Controlada de Monolitos a Microservicios (Horváth, Sakhnenko, & Gurbál, 2024)

Estas estrategias permiten una transición ordenada y reducen los riesgos asociados a la modernización de infraestructuras empresariales, tal como se representa en la Ilustración 14.

3.5.2 Uso de API Gateways para facilitar la comunicación entre componentes

El API Gateway es un componente esencial en la integración de microservicios con sistemas monolíticos, ya que actúa como un punto centralizado de acceso y gestión de solicitudes. Como se describe en la Ilustración 15 su implementación ha sido clave en arquitecturas empresariales híbridas, facilitando la interoperabilidad y garantizando un control eficiente sobre el tráfico de datos (He, 2020). Su función principal es recibir peticiones de los clientes y dirigirlas al servicio correspondiente, proporcionando seguridad, control de tráfico y transformación de datos cuando sea necesario.

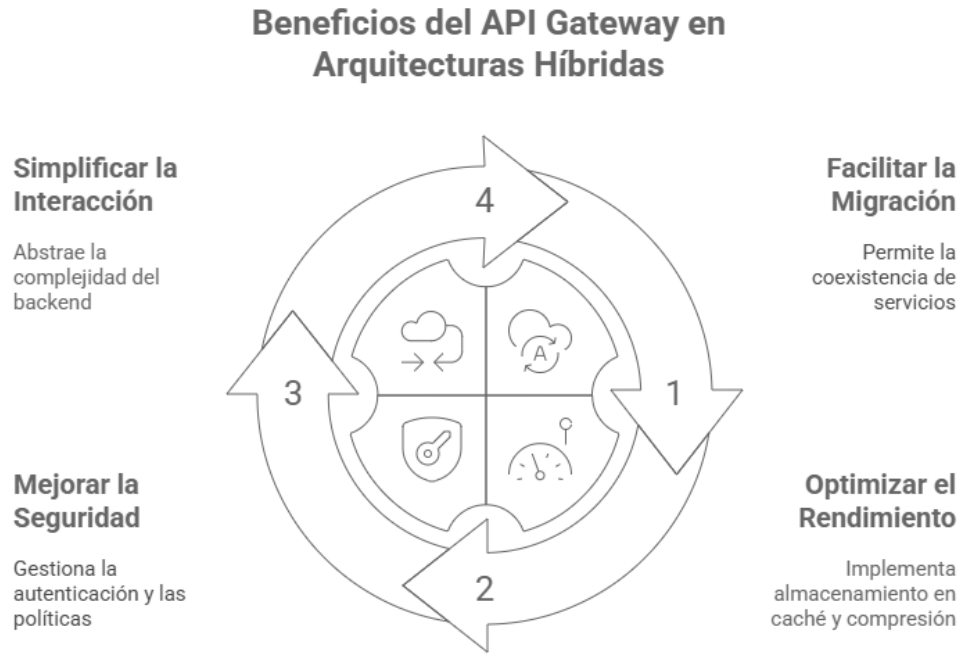

Ilustración 15: Beneficios del API Gateway en Arquitecturas Híbridas (He, 2020)

Ejemplos de herramientas ampliamente utilizadas para la gestión de API Gateways incluyen Kong, Apigee, AWS API Gateway y Nginx. Comparaciones entre estas herramientas han indicado que la elección depende de factores como escalabilidad, seguridad y facilidad de integración con sistemas heredados (Horváth, Sakhnenko, & Gurbál, 2024).

3.5.3 Sincronización de datos entre bases relacionales y sistemas distribuidos

Uno de los desafíos más complejos en la integración de microservicios con sistemas monolíticos es la sincronización de datos entre bases relacionales y entornos distribuidos. Trabajos recientes han abordado soluciones basadas en replicación de datos y eventos distribuidos para minimizar inconsistencias y mejorar la eficiencia de procesamiento (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). En los sistemas monolíticos, la base de datos actúa como un repositorio centralizado, mientras que en los microservicios, cada servicio gestiona su propia base de datos para garantizar autonomía. Las estrategias para manejar esta sincronización incluyen:

- Replicación de datos: La base de datos del sistema monolítico puede replicar información hacia bases de datos distribuidas mediante mecanismos como Change Data Capture (CDC). Esta estrategia ha sido ampliamente adoptada en plataformas con grandes volúmenes de datos para garantizar la coherencia y disponibilidad (Sakhnenko et al., 2024).
- Event Sourcing: Los cambios en los datos pueden registrarse como eventos y ser consumidos por los microservicios según sea necesario.
- APIs de consulta unificada: Permiten que los microservicios consulten información del sistema monolítico sin necesidad de acceso directo a la base de datos.

La elección entre estas estrategias depende de los requerimientos de consistencia y acoplamiento del sistema. Mientras que las APIs unificadas pueden crear un cuello de botella y mantener una dependencia síncrona con el monolito, los patrones basados en eventos como Event Sourcing son a menudo preferidos en arquitecturas modernas. Este enfoque fomenta un desacoplamiento real entre los servicios, permitiendo que operen de forma autónoma. Sin embargo, introduce el concepto de consistencia eventual, un compromiso deliberado que se asume para ganar en escalabilidad y resiliencia, y que es fundamental gestionar en el diseño del sistema.

3.5.4 Retos de la Integración en Arquitecturas Híbridas

La integración de microservicios con sistemas heredados representa un reto significativo para las organizaciones que buscan modernizar sus infraestructuras sin comprometer la estabilidad de sus operaciones. Las dificultades surgen principalmente por la incompatibilidad de paradigmas entre sistemas centralizados y distribuidos (Mishra et al., 2022), (Rahmatulloh et al., 2022). El principal desafío consiste en lograr la compatibilidad entre arquitecturas que fueron diseñadas bajo paradigmas distintos, donde los sistemas monolíticos suelen depender de tecnologías tradicionales con estructuras rígidas, mientras que los microservicios favorecen la modularidad y la independencia de cada servicio. Esta disparidad tecnológica complica la comunicación entre ambos entornos, requiriendo soluciones intermedias como adaptadores, API Gateways y brokers de mensajes que permitan la interacción sin afectar la operatividad del sistema existente.

Otro reto crucial es la gestión de transacciones distribuidas. En un sistema monolítico, las operaciones transaccionales suelen gestionarse a través de una base de datos centralizada, asegurando la coherencia de los datos mediante bloqueos y mecanismos de rollback. Sin embargo, en un entorno de microservicios, cada servicio puede tener su propia base de datos, lo que dificulta la implementación de transacciones de múltiples pasos sin generar inconsistencias. Para mitigar este problema, se han desarrollado patrones como SAGAS, que permiten manejar la compensación de transacciones fallidas a través de eventos distribuidos, minimizando el impacto de fallos en la integridad del sistema y sin comprometer la independencia de los microservicios (Horváth, Sakhnenko, & Gurbál, 2024).

La supervisión y el monitoreo también representan un reto importante en la integración de microservicios con sistemas heredados. Mientras que en un monolito tradicional la depuración de errores y el seguimiento de procesos pueden realizarse de manera centralizada, en una arquitectura distribuida los registros de eventos se dispersan entre múltiples servicios, dificultando la identificación de problemas y su resolución. Para abordar esta problemática, se requieren

herramientas de observabilidad avanzadas, como Prometheus y Jaeger, que permitan rastrear solicitudes en arquitecturas distribuidas y proporcionar visibilidad en tiempo real sobre el estado del sistema para optimizar la detección de fallos (Mishra, Kumar, & Sharma, 2022), (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022).

Finalmente, la validación y la seguridad del sistema híbrido introducen una nueva capa de complejidad. De hecho, la investigación en arquitecturas de componentes distribuidos como los microservicios, que a menudo forman parte de sistemas híbridos, subraya que tanto las pruebas y el aseguramiento de la calidad como la seguridad constituyen desafíos considerables que requieren enfoques evolucionados (Stojanova, Hristoski, Stojanova, & Stojkova, 2023). Las estrategias de pruebas, por ejemplo, deben evolucionar más allá de las pruebas de extremo a extremo (end-to-end), que en estos entornos complejos tienden a volverse frágiles y lentas. Se requieren enfoques como las pruebas de contrato (contract testing) para validar las interacciones entre servicios de forma aislada. En cuanto a la seguridad, donde el perímetro tradicional de la aplicación se desvanece, se vuelve imperativo implementar mecanismos de autenticación y autorización robustos servicio a servicio (por ejemplo, mediante tokens JWT) y gestionar la compleja propagación de la identidad del usuario a través de múltiples saltos en la red, aspectos que reflejan la profundidad de los desafíos de seguridad identificados en la literatura especializada (Stojanova, Hristoski, Stojanova, & Stojkova, 2023).

3.6 Escalabilidad y Alta Concurrencia en Arquitecturas de Microservicios

Las plataformas empresariales actuales deben ser capaces de manejar un alto volumen de solicitudes simultáneas y adaptarse a cambios en la demanda sin degradar su rendimiento. La arquitectura de microservicios permite alcanzar estos objetivos mediante técnicas avanzadas de escalabilidad y gestión de concurrencia.

3.6.1 Uso de contenedores y orquestación en entornos empresariales

El uso de contenedores ha transformado la gestión de aplicaciones empresariales, optimizando el uso de recursos y facilitando la escalabilidad en entornos distribuidos. Avances recientes han demostrado que la adopción de contenedores permite una mejor gestión de cargas de trabajo y una mayor eficiencia en la implementación de arquitecturas de microservicios (Horváth, Sakhnenko, & Gurbál, 2024), (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). La adopción de tecnologías como Docker ha permitido encapsular aplicaciones y sus dependencias en entornos aislados, asegurando su portabilidad y consistencia operativa. Estudios comparativos recientes han demostrado que esta técnica reduce significativamente los errores de configuración y facilita el despliegue continuo en múltiples entornos (He, 2020). En el contexto de microservicios, los contenedores ofrecen una solución ligera y flexible que permite desplegar, actualizar y escalar servicios de manera independiente, reduciendo el impacto de cambios en el sistema global.

La gestión de múltiples contenedores en producción requiere herramientas avanzadas de automatización y coordinación, lo que ha llevado al desarrollo de soluciones como Kubernetes. En entornos de alta concurrencia esta herramienta permite administrar clústeres de contenedores, proporcionando mecanismos avanzados para la distribución de carga, recuperación automática ante fallos, ha sido clave para mejorar la resiliencia y la eficiencia operativa y escalamiento dinámico en función de la demanda (Mishra, Jaiswal, Prakash, & Barwal, 2022). Uno de sus principales beneficios es la capacidad de definir configuraciones declarativas mediante archivos yaml, lo que facilita la estandarización de despliegues en distintos entornos y simplifica la integración con herramientas de monitoreo y seguridad. Esta característica ha sido fundamental en la automatización de procesos y en la reducción del tiempo de despliegue en entornos empresariales (Horváth, Sakhnenko, & Gurbál, 2024).

La implementación de contenedores y su orquestación ofrece beneficios cruciales, destacando especialmente en dos frentes. Primero, resulta fundamental para la

escalabilidad en sistemas empresariales de alta concurrencia, donde la capacidad de ajustar dinámicamente los servicios es clave para el rendimiento. Segundo, la portabilidad que proporcionan es igualmente significativa. Los sistemas basados en contenedores pueden ejecutarse en entornos locales, servidores dedicados o plataformas en la nube sin requerir modificaciones importantes en su configuración. Esta flexibilidad facilita la adopción de estrategias híbridas y multicloud, permitiendo a las organizaciones distribuir sus cargas de trabajo entre distintos proveedores y entornos según sus necesidades de rendimiento, costo y seguridad (Mishra, Jaiswal, Prakash, & Barwal, 2022).

El futuro de los contenedores y su orquestación sigue evolucionando con la adopción de arquitecturas serverless y edge computing, donde la ejecución de microservicios se distribuye de manera aún más eficiente según la proximidad geográfica de los usuarios. Frente a esta situación, Kubernetes sigue consolidándose como una herramienta fundamental para la gestión de aplicaciones escalables, proporcionando una infraestructura resiliente y automatizada (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022) que responde a las necesidades dinámicas del mercado empresarial.

3.6.2 Balanceo de carga y estrategias de distribución de tráfico en microservicios

El balanceo de carga es un componente esencial en la arquitectura de microservicios, ya que permite distribuir el tráfico de manera eficiente entre múltiples instancias de un servicio, garantizando la disponibilidad y el rendimiento del sistema. Diversos ensayos han demostrado que su correcta implementación mejora significativamente la escalabilidad y resiliencia de aplicaciones distribuidas (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022). A medida que las aplicaciones empresariales experimentan un aumento en la demanda, es fundamental implementar estrategias que optimicen el uso de los recursos y eviten la sobrecarga en puntos específicos del sistema.

Como se ejemplifica en la Ilustración 16 una de las principales ventajas del balanceo de carga en entornos de microservicios es la capacidad de distribuir solicitudes en función de métricas en tiempo real, como el nivel de carga de cada nodo, la latencia de respuesta y la proximidad geográfica de los usuarios. Publicaciones recientes han demostrado que estas métricas pueden optimizarse aún más mediante técnicas de aprendizaje automático y predicción de carga (He, 2020), (Horváth, Sakhnenko, & Gurbál, 2024). Para lograr esto, se emplean herramientas especializadas como NGINX, HAProxy y Traefik, que permiten configurar reglas avanzadas para el direccionamiento del tráfico y garantizar un flujo equilibrado de solicitudes.

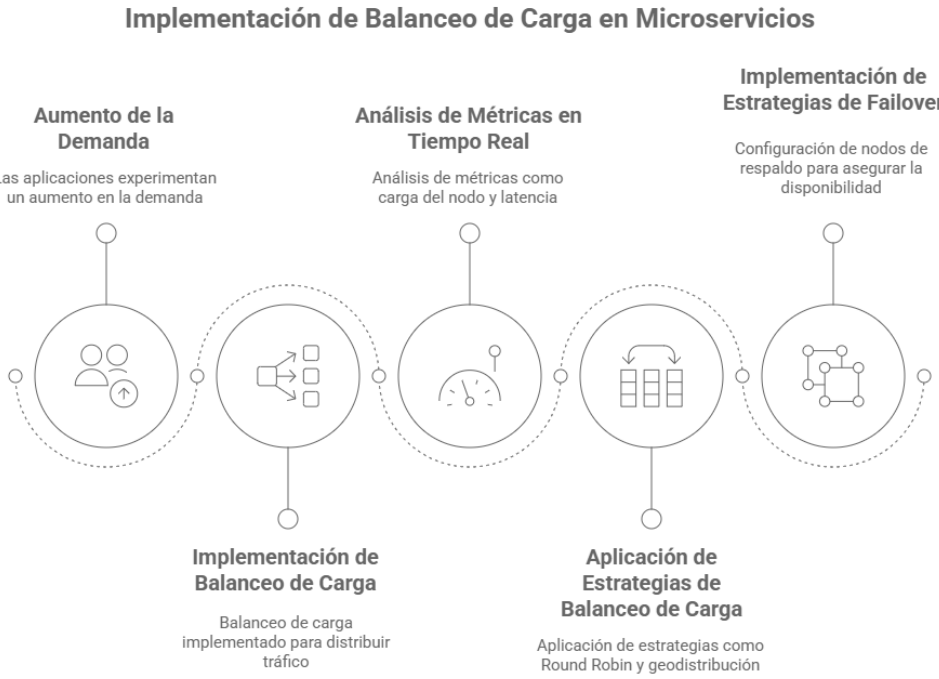

Ilustración 16: Implementación de Balanceo de Carga en Microservicios (Horváth, Sakhnenko, & Gurbál, 2024)

Existen diversas estrategias para la distribución de tráfico en microservicios, cada una con aplicaciones específicas según el tipo de sistema y la infraestructura disponible. Una de las técnicas más utilizadas es el balanceo basado en el algoritmo Round Robin, que asigna las solicitudes de manera equitativa entre todas las instancias disponibles. Sin embargo, en escenarios donde algunas instancias tienen mayor capacidad de procesamiento que otras, se pueden utilizar estrategias como el balanceo basado en peso, que asigna más tráfico a los servidores con mayor

disponibilidad de recursos. Esta práctica reduce la latencia y mejora la eficiencia del procesamiento en entornos distribuidos (Horváth, Sakhnenko, & Gurbál, 2024)

Otra estrategia clave en arquitecturas empresariales es la geodistribución del tráfico, que consiste en redirigir solicitudes a centros de datos cercanos a la ubicación del usuario final. Esta técnica ha sido adoptada en sistemas de alta concurrencia para reducir la latencia en transacciones críticas y garantizar la estabilidad operativa en aplicaciones con demanda global (Sakhnenko et al., 2024).

Además del balanceo de carga, la implementación de estrategias de Failover es crucial para garantizar la disponibilidad del sistema en caso de fallos en una instancia o región específica. La configuración de nodos de respaldo permite que el tráfico sea redirigido automáticamente en caso de interrupciones, evitando la degradación del servicio y asegurando la continuidad operativa.

3.7 Panorama y Casos de Estudio en la Adopción de Microservicios en la Industria

La adopción de arquitecturas de microservicios está transformando la industria, evidenciando mejoras tangibles en escalabilidad, resiliencia y disponibilidad dentro de sectores críticos como el turismo, la hotelería y el transporte. (Barua & Kaiser, 2024) señalan que la modularización de servicios ha permitido a las empresas mejorar la eficiencia operativa y reducir los costos de mantenimiento, mientras que (Laigner, Zhang, Liu, Gomes, & Zhou, 2025) destacan su impacto en la rapidez de despliegue y actualización de aplicaciones. A medida que las aplicaciones empresariales han evolucionado, la demanda de sistemas altamente concurrentes y con baja latencia ha impulsado la adopción de arquitecturas distribuidas. En particular, los sistemas de reservas, que requieren consultas en tiempo real y procesamiento eficiente de grandes volúmenes de datos, se han beneficiado significativamente de la implementación de microservicios y tecnologías de indexación distribuida.

3.7.1 Evolución y Adopción de Arquitecturas de Microservicios en Sistemas de Reservas

Los sistemas de reservas han evolucionado considerablemente con la expansión del comercio electrónico y la digitalización del sector turístico. (Blinowski, Ojdowska, & Przybyłek, 2022) explican que la transición de arquitecturas monolíticas a microservicios ha permitido mejorar la eficiencia en el procesamiento de transacciones y la capacidad de respuesta ante picos de demanda. En sus primeras versiones, estos sistemas estaban diseñados bajo arquitecturas monolíticas, en las que todas las funcionalidades, como la gestión de disponibilidad, reservas, pagos y notificaciones, estaban integradas en una única aplicación. Este enfoque, si bien efectivo en sus inicios, presentaba limitaciones significativas en términos de escalabilidad, mantenimiento y flexibilidad ante cambios en la demanda.

El crecimiento del sector de reservas, en especial con la llegada de plataformas globales como Expedia, Booking.com y Airbnb, evidenció la necesidad de dividir las aplicaciones en módulos más manejables, lo que llevó a la adopción de arquitecturas en capas. Este modelo permitió separar la lógica de negocio de la interfaz de usuario y las bases de datos, facilitando la gestión del código y la actualización de componentes sin afectar el sistema en su totalidad. Sin embargo, las plataformas seguían dependiendo de bases de datos centralizadas y servicios interdependientes, lo que limitaba la escalabilidad en escenarios de alto tráfico.

El auge del cloud computing y las metodologías de desarrollo ágil ha favorecido la adopción de microservicios para superar las limitaciones de los sistemas tradicionales. (Stojanova, Hristoski, Stojanova, & Stojkova, 2023) mencionan que esta migración ha permitido a las plataformas escalar de manera dinámica y optimizar la distribución de carga sin afectar el rendimiento. En este modelo, cada funcionalidad del sistema, como la gestión de usuarios, procesamiento de pagos o consultas de disponibilidad, se encapsula en un servicio independiente con su propio ciclo de vida y base de datos. Esto ha permitido a las plataformas de reservas mejorar su escalabilidad horizontal, implementando estrategias como el autoscaling, que ajusta dinámicamente los recursos según la demanda del sistema.

Otro factor clave en la adopción de microservicios en sistemas de reservas ha sido la necesidad de despliegues continuos y reducción de tiempos de inactividad. A diferencia de los sistemas monolíticos, donde una actualización requiere modificar y probar la aplicación en su conjunto, los microservicios permiten implementar cambios en módulos específicos sin interrumpir el funcionamiento del sistema completo. Esto es particularmente útil en empresas que operan a nivel global, donde cualquier tiempo de inactividad genera pérdidas económicas significativas. Gracias a estos y otros avances, la industria ha logrado optimizar la gestión de tráfico y garantizar la disponibilidad del servicio incluso en picos de alta demanda, como eventos de temporada o promociones globales.

3.7.2 Aplicaciones de microservicios en turismo, hotelería y transporte

El sector turístico ha adoptado arquitecturas de microservicios para mejorar latencia y disponibilidad de sus plataformas. (Tapia, y otros, 2020) describen cómo esta evolución ha permitido a las empresas responder con mayor agilidad a la demanda del mercado sin comprometer la calidad del servicio. Empresas de hotelería, aerolíneas y agencias de viajes han migrado sus sistemas hacia entornos distribuidos para mejorar el rendimiento de sus servicios y garantizar una experiencia de usuario más eficiente.

Airbnb ha estructurado sus servicios en microservicios para gestionar reservas, perfiles de usuario y sistemas de pago de manera independiente. Según (Laigner, Zhang, Liu, Gomes, & Zhou, 2025) esta estrategia ha permitido a la plataforma expandirse globalmente sin comprometer la experiencia del usuario ni el rendimiento de su plataforma, asegurando tiempos de respuesta óptimos incluso en temporadas de alta demanda.

En el sector del transporte, empresas como Uber y Lyft han utilizado arquitecturas de microservicios para gestionar la asignación de conductores, la optimización de rutas y los cálculos de tarifas en tiempo real. Mediante la integración de bases de datos distribuidas y tecnologías de caché, estas plataformas han logrado reducir la

latencia en el procesamiento de información, permitiendo la toma de decisiones automatizada y eficiente.

Compañías como American Airlines y Lufthansa han implementado microservicios para gestionar reservas de vuelos y optimizar la reprogramación de itinerarios. (Tapia, y otros, 2020) explican que, gracias a esta arquitectura, estas aerolíneas han logrado minimizar el impacto de cancelaciones y retrasos mejorando la asignación de recursos en tiempo real. El análisis de estas aplicaciones demuestra que la adopción de microservicios en la industria del turismo, la hotelería y el transporte no solo mejora la escalabilidad y el rendimiento de los sistemas, sino que también facilita la integración con tecnologías emergentes, como la analítica de datos y la inteligencia artificial, permitiendo una mayor personalización en la oferta de servicios.

3.7.3 Evaluación de tecnologías de indexación distribuida en sistemas de alta demanda

La indexación distribuida ha permitido mejorar el rendimiento de los sistemas de alta demanda al facilitar la recuperación eficiente de información en plataformas con millones de consultas simultáneas. (Barua & Kaiser, 2024) destacan que tecnologías como Elasticsearch, Apache Solr y OpenSearch han sido fundamentales para garantizar tiempos de respuesta óptimos en motores de búsqueda empresariales y precisión en la recuperación de datos.

Un caso emblemático de la implementación de indexación distribuida es el de Amazon, que utiliza Elasticsearch para optimizar la búsqueda de productos en su plataforma. Gracias a la capacidad de distribuir índices en múltiples nodos, la empresa ha logrado reducir drásticamente la latencia en la consulta de información, proporcionando resultados relevantes en tiempo real a millones de usuarios en todo el mundo.

Otro ejemplo significativo es el de Netflix, que ha incorporado Apache Solr para mejorar la personalización de recomendaciones mediante la búsqueda semántica y

la categorización eficiente de contenido. Mediante la indexación distribuida, la plataforma ha logrado procesar grandes volúmenes de datos sin comprometer el rendimiento del sistema, asegurando una experiencia fluida para los usuarios.

Booking.com y Expedia han optimizado y mejorado con precisión y rapidez la consulta de disponibilidad de alojamientos y vuelos mediante tecnologías de indexación distribuida. (Blinowski, Ojdowska, & Przybyłek, 2022) resaltan que la implementación de estos motores ha permitido mantener la coherencia de los datos y ofrecer respuestas inmediatas en escenarios de alta concurrencia a sus clientes.

La evaluación de estas tecnologías ha demostrado que la indexación distribuida no solo mejora el rendimiento de los sistemas de alta demanda, sino que también facilita la escalabilidad de las plataformas digitales, permitiendo la gestión eficiente de grandes volúmenes de datos sin afectar la experiencia del usuario.

3.7.4 Casos de estudio en sistemas con alto tráfico

Las plataformas con alto tráfico han integrado estrategias avanzadas de balanceo de carga y almacenamiento en caché para manejar grandes volúmenes de solicitudes y garantizar la disponibilidad de sus servicios en todo momento. (Stojanova, Hristoski, Stojanova, & Stojkova, 2023) explican que la combinación de microservicios y bases de datos distribuidas ha sido clave para mejorar la escalabilidad de estas infraestructuras. Empresas como Google, Facebook y Twitter han desarrollado infraestructuras altamente escalables basadas en microservicios y bases de datos distribuidas para manejar billones de transacciones diarias.

Uno de los casos más destacados es el de Google Search, que emplea tecnologías de indexación distribuida y almacenamiento en caché para procesar miles de millones de búsquedas diarias. Su arquitectura basada en microservicios permite actualizar los índices en tiempo real, asegurando la precisión de los resultados sin comprometer el rendimiento del sistema.

En el sector de las redes sociales, Facebook ha implementado una infraestructura distribuida que combina bases de datos SQL y NoSQL para manejar la interacción de millones de usuarios de manera simultánea. Gracias a la utilización de tecnologías de replicación y almacenamiento en caché, la plataforma ha logrado garantizar la disponibilidad de datos en tiempo real, incluso en momentos de alta demanda.

YouTube maneja uno de los volúmenes de tráfico de video más altos del mundo con una infraestructura basada en microservicios y almacenamiento distribuido. (Laigner, Zhang, Liu, Gomes, & Zhou, 2025) destacan que la integración de almacenamiento en caché y balanceo de carga ha optimizado la entrega de contenido y reducido la latencia en la transmisión de video, asegurando una experiencia fluida para los usuarios sin importar la cantidad de solicitudes simultáneas.

El análisis de estos casos evidencia que la adopción de microservicios, tecnologías de indexación distribuida y almacenamiento en caché es esencial para garantizar el rendimiento de sistemas con alto tráfico. Estas estrategias han permitido a las principales plataformas tecnológicas escalar sus infraestructuras de manera eficiente, asegurando la disponibilidad y la rapidez en la entrega de información.

4. Desarrollo del Proyecto

Este capítulo describe el proceso realizado para analizar la arquitectura monolítica existente, fundamentar la migración hacia un enfoque de microservicios y desarrollar el prototipo que integra técnicas de indexación dinámica y distribuida. Con base en los objetivos planteados, se detalla el diseño de la nueva arquitectura, la configuración de los contenedores y la implementación de un conjunto de microservicios que optimizan la búsqueda de disponibilidad en tiempo real. Adicionalmente, se presentan las validaciones técnicas realizadas para garantizar la escalabilidad y fiabilidad de la solución.

4.1 Análisis del Sistema Existente

4.1.1 Limitaciones de la Arquitectura Monolítica

El sistema de reservas empresarial actualmente opera bajo una arquitectura monolítica que integra la lógica de negocio, la gestión de datos y la interfaz de usuario en una única aplicación desplegable. Esta estructura, aunque funcional en etapas iniciales, ha evidenciado limitaciones críticas conforme la plataforma ha crecido en volumen de usuarios y operaciones concurrentes.

Durante periodos de alta demanda, se han registrado tiempos promedio de respuesta superiores a los 15 segundos al consultar la disponibilidad de espacios desde la interfaz web, especialmente al ejecutar consultas sobre la tabla de base de datos que gestiona dicha información, ya sea con o sin filtros por fecha, categoría y estado. Este retardo ha afectado directamente la experiencia del usuario y los indicadores de conversión.

Los principales cuellos de botella se han identificado en la capa de acceso a datos, donde múltiples consultas concurrentes compiten por acceder a la base de datos relacional principal. Además, el acoplamiento estricto entre módulos imposibilita la

realización de despliegues parciales. Por ejemplo, para actualizar el módulo de reportes, es necesario reiniciar toda la aplicación, lo cual genera ventanas de inactividad no deseadas y un mayor riesgo operativo. Esta situación limita la agilidad del equipo de desarrollo para introducir mejoras incrementales.

Por último, cualquier intento de escalar el sistema requiere replicar por completo la aplicación monolítica, lo que incrementa el consumo de recursos de manera no proporcional a la demanda real. Esta estrategia representa un uso ineficiente de la infraestructura, ya que se asignan más recursos a componentes que no lo requieren, como los módulos administrativos, cuya carga es predominantemente estática, mientras que los procesos intensivos, como las búsquedas, continúan saturando los mismos puntos críticos.

4.1.2 Identificación de Procesos Críticos

Basado en la experiencia adquirida durante la gestión del sistema monolítico y en el análisis de la documentación técnica disponible, se identificaron los siguientes procesos como críticos para el rendimiento y la experiencia del usuario:

1. Búsqueda de disponibilidad: El sistema debe procesar un alto volumen de consultas simultáneas para verificar cupos, espacios o reservas disponibles.
2. Actualización de reservas: La confirmación o cancelación de reservas genera actualizaciones frecuentes en la base de datos central.
3. Procesamiento de pagos y verificación de transacciones: Aunque no forma parte central de la búsqueda de disponibilidad, resulta sensible a la carga y requiere integridad en las operaciones.

La naturaleza y exigencia de estos procesos evidencian la necesidad de una arquitectura capaz de responder de forma inmediata a las solicitudes de los clientes, lo que no solo refuerza la urgencia de migrar hacia un enfoque basado en microservicios e indexación dinámica, sino que también apunta a mejorar significativamente la experiencia del usuario en escenarios de alta demanda.

4.1.3 Justificación de la Migración hacia Microservicios

La migración hacia una arquitectura de microservicios responde a la necesidad de superar las limitaciones críticas de rendimiento y escalabilidad del sistema monolítico actual. Más que una simple modernización, se trata de una estrategia para eliminar cuellos de botella y la rigidez estructural que afectan tanto la experiencia del usuario como la agilidad operativa del negocio.

Frente a latencias superiores a 15 segundos y la imposibilidad de escalar componentes de forma selectiva, se justifica fragmentar el sistema. Al aislar la lógica de búsqueda en un microservicio autónomo, se aborda la raíz del problema: la sobrecarga de la base de datos y el fuerte acoplamiento entre módulos. Esto permite optimizar y escalar la funcionalidad más crítica sin replicar toda la aplicación.

Por lo tanto, la transición se justifica porque aborda directamente las deficiencias identificadas:

- Resuelve el cuello de botella en el rendimiento: Desacopla las consultas de alta demanda del núcleo monolítico, delegándolas a un servicio especializado que utiliza indexación distribuida para ofrecer respuestas en tiempo real.
- Permite una escalabilidad eficiente y económica: Habilita la réplica individual de los servicios que enfrentan mayor carga, evitando el costo y el uso ineficiente de recursos que implica escalar toda la aplicación monolítica.
- Aumenta la agilidad del desarrollo: Facilita la introducción de mejoras y actualizaciones en componentes aislados, superando la lentitud y el riesgo operativo asociados al ciclo de despliegue del monolito.

La migración es una medida correctiva y evolutiva, diseñada para alinear la infraestructura tecnológica del sistema de reservas con las exigencias de inmediatez, disponibilidad y flexibilidad del mercado actual.

4.2 Diseño de la Arquitectura Propuesta

4.2.1 Principios de Diseño Adoptados

El diseño de la nueva arquitectura se fundamenta en los siguientes pilares estratégicos, orientados a resolver las limitaciones del sistema actual y asegurar una evolución sostenible:

- Dominio y Autonomía de Servicios: Cada microservicio se diseña en torno a una capacidad de negocio específica (un dominio), siendo responsable de su propia lógica y persistencia de datos. Este principio de alta cohesión y bajo acoplamiento es fundamental para permitir el desarrollo, despliegue y escalado independiente de cada componente.
- Segregación de Responsabilidades de Consulta (CQRS): Para lograr un rendimiento óptimo, se separa el modelo de lectura (consultas de disponibilidad) del de escritura (actualizaciones de espacios disponibles). Las consultas se delegan a un modelo de lectura altamente optimizado mediante un motor de indexación distribuida (Elasticsearch), liberando así a la base de datos relacional del monolito de la carga de lectura intensiva.
- Comunicación Asíncrona y Desacoplada: La interacción entre el sistema monolítico y los nuevos microservicios se basa en una arquitectura orientada a eventos, utilizando un bróker de mensajes (RabbitMQ). Este enfoque elimina las dependencias síncronas y rígidas, aumentando la resiliencia y la capacidad de respuesta del ecosistema completo.
- Coexistencia y Migración Progresiva: La arquitectura está diseñada para coexistir con el sistema existente, garantizando la continuidad operativa y minimizando los riesgos del cambio. Se adopta el patrón Strangler (Ilustración 3), permitiendo una transición controlada donde los nuevos servicios reemplazan gradualmente la funcionalidad del monolito.

Estos principios permiten diseñar una arquitectura que optimice las búsquedas en tiempo real sin comprometer la estabilidad ni el funcionamiento de los procesos críticos del sistema monolítico.

4.2.2 Descomposición por Capacidades de Negocio (Domain-Driven Design)

La descomposición del sistema se realizó siguiendo los principios de Domain-Driven Design (DDD), identificando las capacidades de negocio críticas que debían ser aisladas del monolito. Este enfoque permite definir microservicios con fronteras claras y responsabilidades únicas, conocidos como Contextos Delimitados.

Para la gestión de la disponibilidad en tiempo real, se definieron los siguientes microservicios:

Servicio de Consulta de Disponibilidad (Read Model):

- Responsabilidad: Actúa como el modelo de lectura optimizado del sistema. Su única función es responder a las consultas de disponibilidad de los clientes de la manera más rápida posible.
- Funcionamiento: Expone una API REST pública y opera exclusivamente sobre los datos pre-indexados en Elasticsearch y cacheados en Redis. No tiene conocimiento de la lógica de negocio para crear o modificar la disponibilidad, lo que garantiza su simplicidad y alto rendimiento.

Servicio de Gestión de Disponibilidad (Write Model / Synchronization Service):

- Responsabilidad: Es el responsable de mantener la consistencia del modelo de lectura. Su función es crear, modificar y eliminar los registros de disponibilidad.
- Funcionamiento: Este servicio no expone una API pública. En su lugar, escucha eventos provenientes del sistema monolítico (a través de RabbitMQ) cada vez que ocurre un cambio (ej. una nueva reserva). Al recibir un evento, procesa la información y actualiza el índice en Elasticsearch para reflejar el estado más reciente, asegurando la consistencia eventual de los datos.

La definición de estos microservicios, uno como el modelo de consulta y otro como el gestor de datos, materializa el principio de CQRS y permite desacoplar de manera efectiva las operaciones.

4.2.3 Selección de Tecnologías

La selección de tecnologías se basó en los requisitos de la arquitectura propuesta, priorizando el rendimiento en tiempo real, la escalabilidad, la resiliencia y la facilidad de integración. El stack tecnológico se compone de las siguientes herramientas clave:

- Motor de Indexación y Búsqueda: Se seleccionó Elasticsearch como motor principal para la indexación distribuida por su capacidad para ejecutar búsquedas complejas sobre grandes volúmenes de datos con latencias de milisegundos. Su arquitectura nativamente distribuida y su potente API de consulta fueron decisivas para implementar el modelo de lectura (Read Model) de manera eficiente.
- Capa de Caché en Memoria: Se implementó Redis como una capa de caché distribuido para almacenar los resultados de las consultas más frecuentes, gracias a su altísimo rendimiento en operaciones de lectura/escritura y por su soporte para estructuras de datos avanzadas y políticas de expiración (TTL). Su rol es reducir la carga sobre Elasticsearch y entregar respuestas de forma casi instantánea para solicitudes repetidas.
- Bróker de Mensajería: Se optó por RabbitMQ para gestionar la comunicación asíncrona y basada en eventos entre el monolito y los microservicios. Su fiabilidad, flexibilidad en el enrutamiento de mensajes y su robusto soporte para patrones como publish/subscribe, características que lo convierten en la opción ideal para garantizar una sincronización de datos desacoplada y resiliente.
- Contenerización y Orquestación Local: El ecosistema de microservicios se desplegó en contenedores Docker, utilizando Docker Compose para definir y ejecutar la aplicación multi-contenedor en un entorno de desarrollo y despliegue local.

- API Gateway y Proxy Inverso: Se seleccionó Nginx como API Gateway por su alto rendimiento, estabilidad y su potente módulo de reescritura de rutas. Su función es actuar como el punto único de entrada, gestionando la seguridad básica, el balanceo de carga y, fundamentalmente, el enrutamiento de solicitudes entre el sistema monolítico y los nuevos microservicios, siendo una pieza clave en la implementación del patrón Strangler.

La Ilustración 17 resume visualmente este stack tecnológico, destacando el propósito fundamental que cada herramienta cumple dentro de la arquitectura diseñada.

Ilustración 17: Tecnologías Seleccionadas

4.2.4 Modelo de Integración y Comunicación con el Monolito

La estrategia de comunicación se diseñó para cumplir un objetivo principal: integrar la nueva funcionalidad de búsqueda de alto rendimiento sin alterar el núcleo del sistema monolítico existente, garantizando una transición transparente para los clientes y una total continuidad operativa. Para lograrlo, se implementó un modelo dual que separa el flujo de solicitudes de clientes (síncrono) del flujo de actualización de datos (asíncrono).

Gestión de solicitudes a través del API Gateway

Se introduce un API Gateway como fachada y punto de entrada único para todas las peticiones del sistema. Este componente es crucial para la coexistencia de ambas arquitecturas:

- Enrutamiento Transparente: El Gateway recibe todas las solicitudes. Por defecto, las redirige sin cambios al componente REST existente del monolito, que sigue gestionando la gran mayoría de operaciones de lectura y escritura del negocio.
- Intercepción y Desvío (Patrón Strangler): Solo las solicitudes específicas para la búsqueda de disponibilidad son interceptadas por el Gateway y enrutadas hacia el nuevo Microservicio de Búsqueda de Disponibilidad. De esta manera, se reemplaza una funcionalidad crítica sin afectar al resto del sistema.

Sincronización de Datos mediante una Arquitectura de Eventos

Para que el nuevo módulo de búsqueda opere con datos actualizados, el monolito participa activamente en la sincronización a través de un mecanismo de eventos desacoplado:

- Productor de Eventos en el Monolito: Dentro del monolito, se integra un módulo adaptador. Su única responsabilidad es capturar los cambios de estado relevantes (una reserva confirmada o cancelada por ejemplo),

traducir esos datos al formato requerido por los nuevos servicios y publicarlos como un mensaje en un bróker RabbitMQ.

- Microservicios Consumidores: Del lado de los consumidores, el Microservicio de Gestión de Disponibilidad está suscrito a RabbitMQ para procesar los eventos. Al recibir un mensaje, este servicio orquesta una secuencia de acciones: primero, actualiza el índice correspondiente en Elasticsearch para reflejar el nuevo estado de la disponibilidad y, acto seguido, emite un comando para invalidar la entrada relevante en la caché de Redis. Este proceso coordinado asegura que tanto la fuente de búsqueda principal como los datos cacheados se mantengan sincronizados con los cambios originados en el monolito.

En definitiva, este modelo de integración y comunicación es la clave para alcanzar los objetivos del proyecto. Se logra modernizar la funcionalidad crítica de búsqueda sin desestabilizar el sistema legado, se garantiza la autonomía de los componentes y se establece un flujo de datos resiliente basado en la consistencia eventual, tal como se detalla en la Ilustración 18.

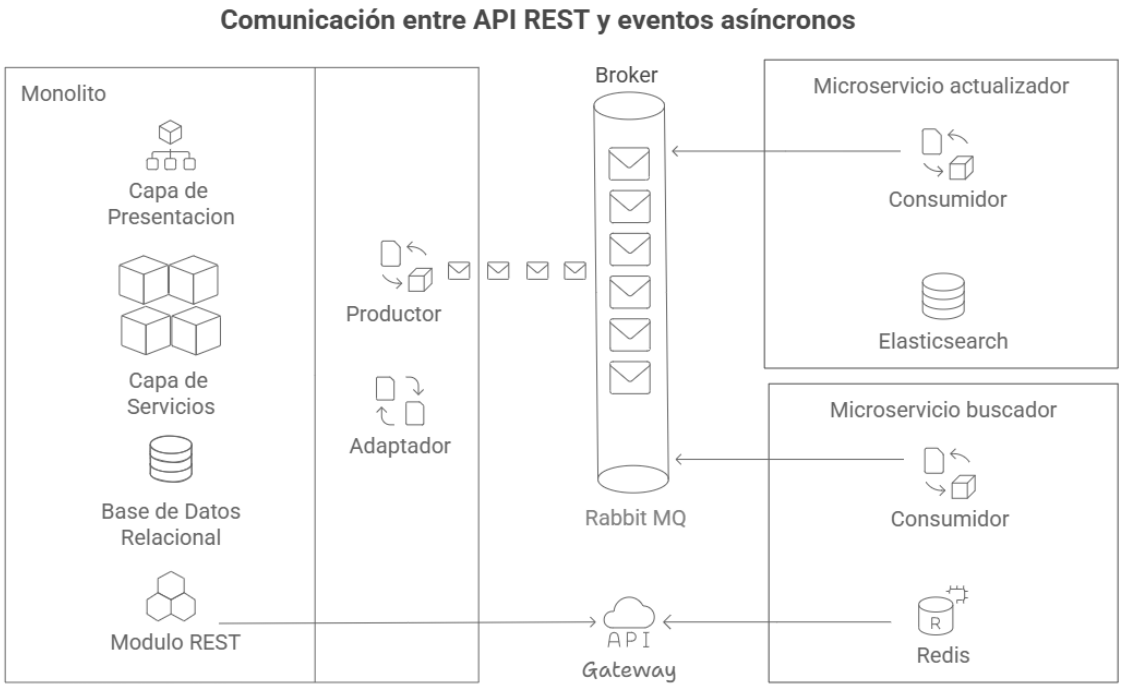

Ilustración 18: Comunicación entre API REST y eventos asíncronos

4.2.5 Seguridad y Gestión de Autenticación

La seguridad del sistema se implementa bajo un enfoque de defensa en profundidad, orientado a proteger tanto el acceso externo como la comunicación interna entre servicios. Se contemplan los siguientes mecanismos:

- Autenticación Centralizada en el API Gateway: Como primera línea de defensa, todas las solicitudes externas deben pasar por el API Gateway. Este componente se encarga de la autenticación, verificando la firma y la validez de los tokens JWT (JSON Web Tokens) presentados por el cliente. Al centralizar esta validación, se establece un punto de control único y se desacopla la lógica de autenticación del resto de los servicios.
- Autorización Distribuida en Microservicios: Una vez que una solicitud es autenticada por el Gateway, cada microservicio realiza una segunda verificación. No solo re-valida la integridad del token, sino que ejecuta la autorización, es decir, inspecciona los claims del token (como el rol o los permisos del usuario) para decidir si la operación solicitada está permitida. Esto evita accesos no autorizados incluso si la solicitud ya ha pasado el primer filtro.
- Cifrado de Tráfico en Tránsito (SSL/TLS): Toda la comunicación entre los componentes del sistema (cliente-Gateway, Gateway-microservicios, microservicio-monolito) se cifra mediante certificados SSL/TLS. Esto protege la integridad y confidencialidad de los datos mientras viajan por la red, mitigando riesgos de ataques como man-in-the-middle.

Esta estrategia de seguridad multicapa es fundamental para la viabilidad de la arquitectura propuesta. Al adoptar un modelo de "confianza cero" (Zero Trust), donde las solicitudes se validan en cada punto crítico, se obtiene un sistema resiliente a amenazas tanto externas como internas. Se logra así un equilibrio pragmático entre una seguridad férrea, la escalabilidad horizontal de los servicios y la facilidad de mantenimiento, lo que valida el diseño no solo para el prototipo, sino también para una eventual puesta en producción. El flujo de validación de tokens en el Gateway y los microservicios se visualiza en la Ilustración 19.

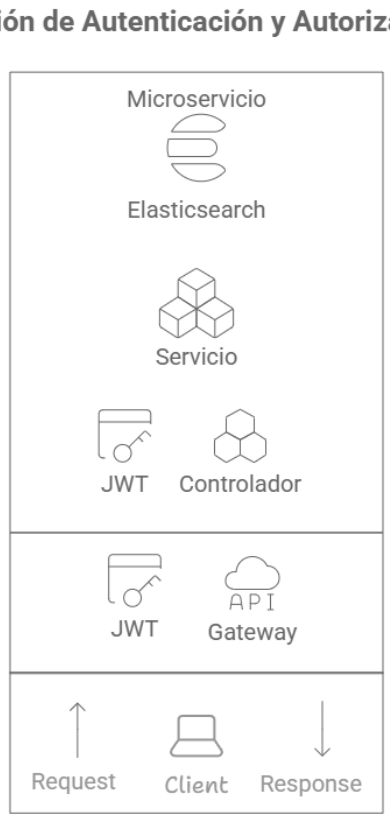

Ilustración 19: Gestión de Autenticación y Autorización

4.3 Arquitectura Física y Lógica del Sistema

4.3.1 Arquitectura Lógica de Microservicios

La arquitectura lógica propuesta organiza los componentes en torno a dos dominios funcionales claramente diferenciados, que se corresponden directamente con los microservicios definidos previamente. Cada dominio tiene responsabilidades específicas sobre la gestión y exposición de la disponibilidad:

- Dominio de Consulta (Read Model): Este dominio está materializado por el Servicio de Consulta de Disponibilidad. Su única responsabilidad es atender las peticiones de búsqueda de los clientes. Opera sobre una vista de datos altamente optimizada y previamente indexada en Elasticsearch, con una capa de caché en Redis para acelerar las respuestas recurrentes. Su diseño está enfocado exclusivamente en la velocidad y eficiencia de la lectura.

- Dominio de Gestión (Write Model): Este dominio lo conforma el Servicio de Gestión de Disponibilidad. Se encarga de mantener la integridad y actualidad del modelo de lectura. Su ciclo de operación se activa por eventos: procesa los mensajes de cambio provenientes del monolito, sincroniza los datos con el índice de Elasticsearch y gestiona la invalidación de la caché, asegurando la consistencia eventual del sistema.

Esta separación de dominios no solo facilita la especialización y escalabilidad de cada componente, sino que también permite mantener una arquitectura orientada a la responsabilidad única, donde las operaciones de lectura y escritura se gestionan de manera desacoplada y eficiente. Esto facilita la incorporación de nuevas capacidades sin comprometer la estabilidad del monolito ni aumentar innecesariamente la complejidad del prototipo.

4.3.2 Arquitectura de Red y Topología del Despliegue

Para el despliegue del prototipo, se implementó una topología distribuida que utiliza Servidores Privados Virtuales (VPS) independientes. Cada VPS aloja un conjunto de servicios agrupados por su dominio funcional, los cuales son definidos y gestionados como contenedores mediante Docker Compose.

La comunicación se gestiona de la siguiente manera:

- La comunicación interna entre los microservicios se realiza de forma segura y eficiente a través de una red virtual privada gestionada por Docker.
- El tráfico externo se canaliza a través del API Gateway (Nginx), que actúa como punto de entrada unificado. Este se encarga de redirigir las solicitudes hacia los microservicios o al sistema monolítico según corresponda, como se definió en el modelo de comunicación.

Esta topología de despliegue se visualiza en la Ilustración 20.

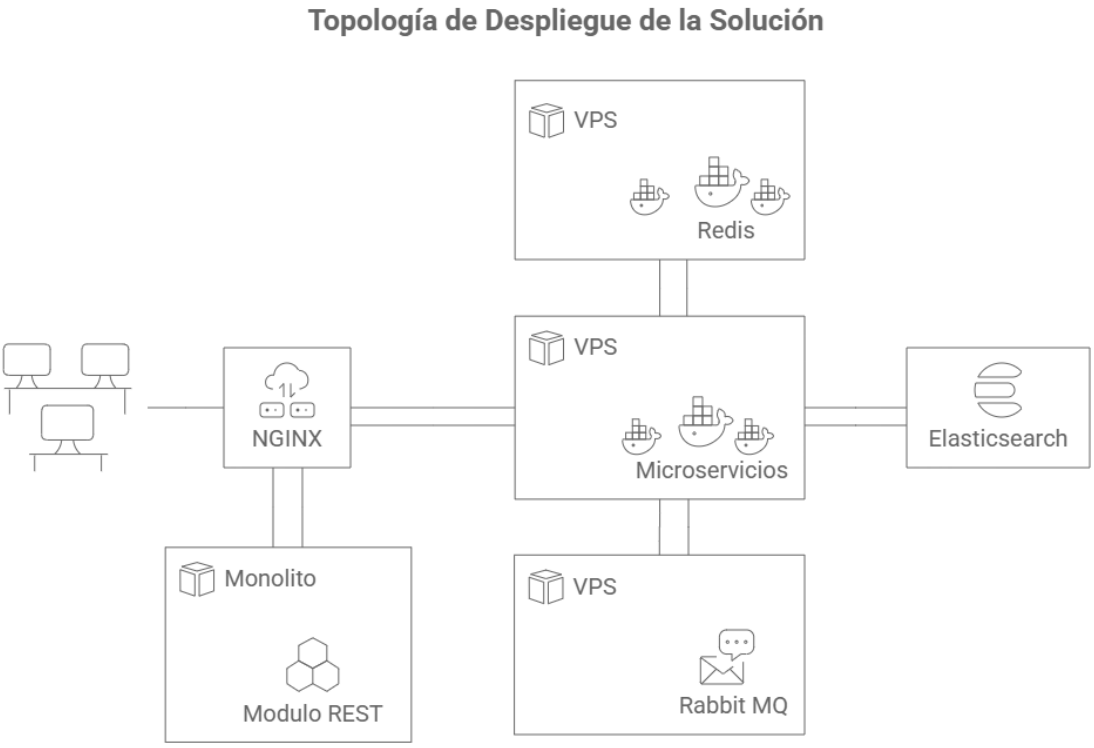

Ilustración 20:Topología de Despliegue de la Solución

La arquitectura de red y la topología descritas se traducen en varios beneficios clave para el sistema:

- Facilita la escalabilidad controlada: Al separar los servicios en VPS distintos, es posible escalar componentes críticos de forma independiente según la demanda observada durante las pruebas.
- Mejora la disponibilidad: La segmentación por nodos y la contenerización permiten reiniciar o redistribuir servicios rápidamente en caso de fallos, minimizando el impacto en el resto del sistema.
- Habilita la aplicación del patrón Strangler: Mediante reglas de enrutamiento en el API Gateway, se redirige progresivamente el tráfico hacia los microservicios, facilitando una migración funcional y controlada.

4.3.3 Configuración de Contenedores y Coordinación de Servicios

La gestión del ciclo de vida de los microservicios se basa en una estrategia de contenerización para asegurar la portabilidad y consistencia del entorno.

Construcción de Imágenes Docker:

Para cada microservicio, se construyó una imagen Docker independiente y auto-contenida. Cada imagen empaqueta el código de la aplicación, su runtime específico y todas las dependencias necesarias para su funcionamiento, incluyendo las librerías para interactuar con servicios externos como Elasticsearch, Redis y el sistema monolítico. Este enfoque garantiza que cada servicio se ejecute de la misma manera en cualquier entorno.

Coordinación con Docker Compose:

El despliegue y la coordinación de los servicios se orquestan mediante un único archivo docker-compose.yml. Este archivo define de manera declarativa toda la pila de la aplicación, permitiendo levantar y conectar todos los contenedores con un solo comando. Esta configuración proporciona beneficios clave para la gestión del prototipo:

- Despliegue Controlado y Reproducible: Cada componente (microservicios, Nginx, RabbitMQ, etc.) se define como un "servicio" dentro del archivo, lo que facilita su ejecución, reinicio y mantenimiento de forma predecible y sin necesidad de herramientas de orquestación más complejas.
- Comunicación Interna Simplificada: Docker Compose crea una red virtual para los servicios. Esto permite que se comuniquen entre sí utilizando sus nombres de servicio como si fueran nombres de host (DNS), eliminando la necesidad de gestionar direcciones IP específicas.
- Procesamiento Desacoplado: Si bien no se implementa un balanceador de carga activo entre instancias, la arquitectura de eventos permite que múltiples réplicas de un microservicio consumidor puedan procesar mensajes de la cola de forma independiente, logrando un escalado de procesamiento eficiente y resiliente.

Este enfoque mantiene la simplicidad necesaria para el prototipo, al tiempo que permite validar la viabilidad técnica de una arquitectura basada en contenedores y servicios distribuidos.

4.4 Implementación del Prototipo

4.4.1 Configuración del Flujo de Trabajo y Entorno de Desarrollo

Para la implementación del prototipo, se estableció un flujo de trabajo profesional y automatizado, diseñado para gestionar la complejidad inherente de una arquitectura distribuida y garantizar la máxima calidad, consistencia y eficiencia durante todo el ciclo de desarrollo. Este flujo metodológico se sustentó en tres pilares fundamentales:

1. Entorno de Desarrollo Local Consistente

El desarrollo local se basó en Docker Compose. Este enfoque permitió replicar la arquitectura de producción de manera exacta en el entorno de desarrollador. Al definir la pila completa de servicios (microservicios, Elasticsearch, Redis, RabbitMQ) y sus redes en un único archivo docker-compose.yml, se erradicaron los típicos problemas de inconsistencia entre máquinas (el conocido "en mi máquina sí funciona"), garantizando así que el prototipo se desarrollara sobre una base estable, homogénea y predecible.

2. Estrategia de Control de Versiones con Git

Se utilizó Git como sistema de control de versiones, con Bitbucket como plataforma centralizada para los repositorios. Se adoptó una estrategia de repositorios individuales por microservicio, lo que fomenta la autonomía y el versionado independiente de cada componente. Para la gestión de ramas, se siguió un modelo basado en GitFlow, utilizando ramas feature para nuevas funcionalidades, develop para la integración y main para las versiones estables, permitiendo un desarrollo ordenado y revisiones de código (pull requests) antes de cada integración.

3. Pipeline de Integración Continua (CI/CD) con Jenkins

Para automatizar el proceso de validación y despliegue del prototipo, se implementó un pipeline de CI/CD para cada microservicio utilizando un script en formato declarativo. Este enfoque permite versionar la lógica de automatización junto con el código fuente y asegura un proceso estandarizado y repetible.

El pipeline se estructura en las siguientes etapas secuenciales:

1. Clonación del Repositorio (Cloning Git): El proceso se inicia con la clonación del código fuente desde su repositorio dedicado en Bitbucket, asegurando que se trabaje siempre sobre la última versión de la rama de desarrollo (develop).
2. Compilación Aislada con Maven (Build Spring Boot Project): Para garantizar un entorno de compilación limpio y reproducible, la compilación del proyecto Spring Boot no se ejecuta directamente en el agente de Jenkins. En su lugar, se utiliza un enfoque de "build-in-Docker": se levanta un contenedor efímero con la imagen de Maven, se monta el código fuente y se ejecuta el comando mvn clean package. Esta técnica aísla el proceso de compilación de la configuración de la máquina anfitriona.
3. Ejecución en un Entorno Contenerizado (Run Spring Boot App): Una vez compilado el artefacto (.jar), el pipeline realiza el despliegue del prototipo. Primero, detiene y elimina cualquier versión anterior del contenedor del microservicio. Acto seguido, ejecuta un nuevo contenedor a partir de una imagen base de Java (Eclipse Temurin JRE), montando el archivo .jar recién generado y exponiendo el puerto correspondiente.
4. Notificación de Estado (post): Finalmente, el pipeline notifica el resultado de la ejecución. Informa si el proceso fue exitoso, indicando que la nueva versión del servicio está en ejecución, o si falló, facilitando la detección de errores.

La ejecución exitosa de este pipeline se puede observar en la Ilustración 21, donde se aprecian las distintas etapas completadas satisfactoriamente en la interfaz de Jenkins.

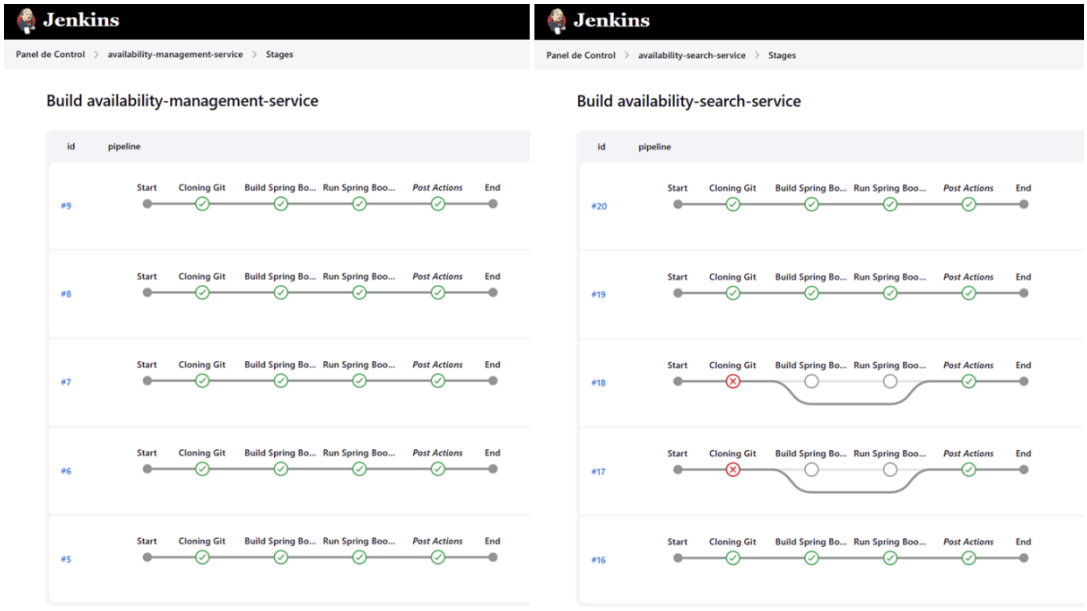

Ilustración 21: Flujo de Ejecución del Pipeline de CI/CD en Jenkins

4.4.2 Implementación Técnica de los Microservicios

La materialización de la arquitectura propuesta se llevó a cabo utilizando un stack tecnológico basado en Java y el framework Spring Boot. Esta elección se fundamentó en la madurez del ecosistema, su robusto soporte para la creación de microservicios y la facilidad para integrarse con las tecnologías seleccionadas como Elasticsearch, Redis y RabbitMQ.

Implementación del Microservicio de Consulta (Read Model), este servicio, enfocado en la velocidad de lectura, se construyó con los siguientes componentes principales de Spring Boot:

- API REST: Se utilizó spring-boot-starter-web para crear los endpoints (@RestController) que exponen la funcionalidad de búsqueda. Estos reciben las peticiones del cliente, las procesan y devuelven la disponibilidad en formato JSON.
- Conectividad con la Caché: Mediante spring-boot-starter-data-redis, se implementó la lógica para interactuar con Redis. Antes de consultar el motor de búsqueda, el servicio verifica si el resultado para una consulta idéntica ya existe en la caché, retornándolo de inmediato si es el caso.

- Cliente de Búsqueda: La comunicación con Elasticsearch se gestionó a través de su cliente Java, permitiendo construir y ejecutar consultas complejas de manera programática sobre los índices de disponibilidad.

Implementación del Microservicio de Gestión (Write Model): La principal responsabilidad de este servicio es procesar eventos de forma asíncrona. Su implementación se centró en:

- Consumidor de Eventos: Se utilizó spring-boot-starter-amqp para configurar un oyente (@RabbitListener) que se suscribe a la cola de RabbitMQ. Este componente se activa automáticamente cada vez que el monolito publica un nuevo mensaje de cambio.
- Lógica de Sincronización: Una vez que un evento es consumido, la lógica de negocio del servicio se encarga de transformar los datos del mensaje y utilizar el cliente de Elasticsearch para actualizar (crear, modificar o eliminar) los documentos en el índice correspondiente, manteniendo así la consistencia del modelo de lectura.

El uso del ecosistema Spring Boot permitió implementar los patrones de diseño definidos (CQRS, comunicación por eventos) de una manera estructurada y eficiente, acelerando el desarrollo del prototipo y asegurando una base de código mantenible y escalable.

4.4.3 Implementación del API Gateway

Para orquestar el flujo de solicitudes y actuar como la fachada del sistema, se implementó un API Gateway basado en Nginx. Este componente es el punto de entrada único para todo el tráfico externo y su configuración fue crucial para la integración transparente del monolito con los nuevos microservicios.

Su implementación cumple con las siguientes funciones clave:

- Redirección de Tráfico (Strangler Pattern): Se configuraron reglas de enrutamiento específicas. La mayoría de las rutas (location) se dirigen por

defecto al sistema monolítico. Sin embargo, las rutas correspondientes a la nueva funcionalidad, como …/webservice/agencyServices/findAvailability, son interceptadas y redirigidas al microservicio de búsqueda correspondiente.

- Centralización de la Seguridad: El Gateway actúa como la primera barrera de seguridad, encargándose de validar los tokens JWT de las solicitudes entrantes antes de pasarlas a cualquier servicio interno.
- Enrutamiento Transparente y Abstracción: Para los sistemas cliente, la complejidad del backend es invisible. Interactúan con un único dominio y una API unificada, mientras que el Gateway gestiona internamente la complejidad de saber qué componente (monolito o microservicio) debe responder.
- Facilitador de la Migración: Al ser el punto de control del tráfico, el Gateway es la pieza que permite la evolución del sistema. Nuevas rutas pueden ser añadidas o modificadas para apuntar a nuevos microservicios de forma gradual, sin interrumpir el servicio.

Estas funciones, que son cruciales para la coexistencia de ambas arquitecturas, se resumen en la Ilustración 22.

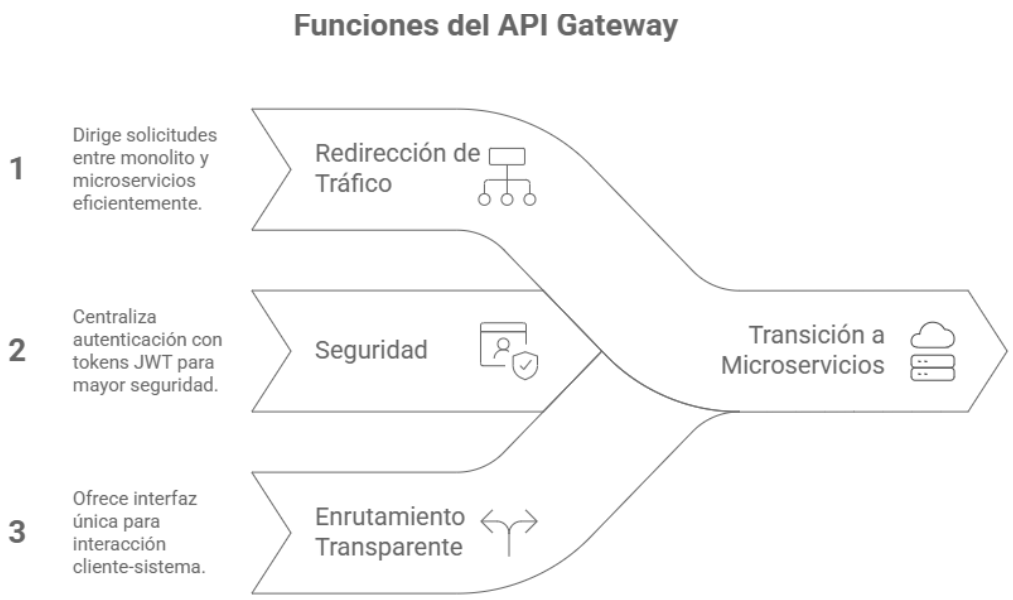

Ilustración 22: Funciones del API Gateway

La configuración del Gateway se define mediante archivos de configuración versionables, lo que facilita tanto la incorporación de nuevas rutas como la desactivación progresiva de componentes monolíticos conforme se avanza en el proceso de migración.

4.4.4 Implementación de la Integración con el Sistema Monolítico

La integración entre los nuevos microservicios y el sistema monolítico se implementó con un enfoque pragmático, priorizando la estabilidad y la no alteración del código legado. Como se representa en la Ilustración 23, el objetivo fue equilibrar la continuidad del monolito con la innovación de la nueva arquitectura. Esto se logró a través de los siguientes puntos de implementación técnica:

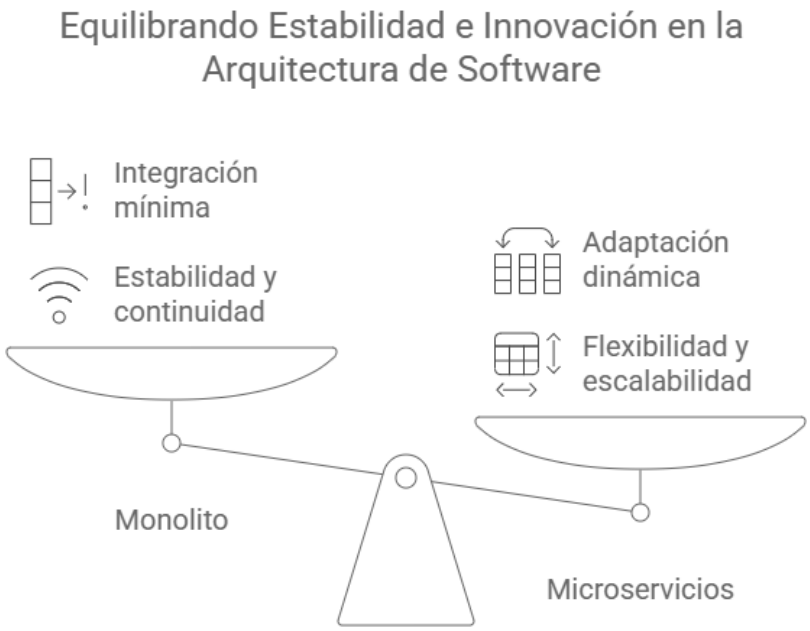

Ilustración 23: Equilibrando Estabilidad e Innovación en la Arquitectura de Software

- Exposición Controlada de Endpoints REST: En el monolito, se configuró un conjunto mínimo y específico de endpoints REST. Estos no son de acceso público, sino internos, y están securizados para ser consumidos únicamente por los microservicios autorizados. Su función principal es de solo lectura,

proveyendo a los nuevos servicios una vía controlada para consultar el estado actual de las reservas cuando sea estrictamente necesario.

- Implementación del Módulo Productor de Eventos: Se desarrolló un nuevo módulo interno en el monolito, diseñado para "engancharse" a la lógica de negocio existente. Tras una operación de base de datos que modifica la disponibilidad (ejemplo una transacción de reserva confirmada), este módulo se activa para:

1. Construir un mensaje con los datos relevantes del cambio.
2. Utilizar una librería cliente de AMQP para publicar dicho mensaje de forma asíncrona en RabbitMQ. Esta implementación se realizó de manera que no afectara el rendimiento de la transacción original.

- Contrato de Datos mediante DTOs: Para la comunicación entre ambos sistemas, se implementaron Objetos de Transferencia de Datos (DTOs). Estas clases Java simples actúan como un contrato formal y estable. El módulo adaptador en el monolito se encarga de mapear las entidades internas del sistema a estos DTOs antes de enviarlos, y los microservicios hacen lo inverso. Este patrón asegura que los sistemas estén desacoplados a nivel de modelo de datos: la estructura interna del monolito puede cambiar sin romper a los microservicios, y viceversa.

Estas medidas técnicas permitieron mantener una interoperabilidad segura y eficiente entre ambas arquitecturas, asegurando que los procesos relacionados con la disponibilidad se mantengan consistentes y sincronizados sin comprometer la operatividad del sistema principal.

5. Resultados y discusión

5.1 Metodología de pruebas de carga y criterios de evaluación

Con el objetivo de validar el comportamiento del prototipo propuesto bajo condiciones de uso intensivo, se diseñó una serie de pruebas de carga orientadas a medir la capacidad de respuesta, la tolerancia a fallos y el rendimiento general del sistema en distintos escenarios. Estas pruebas se centraron en el endpoint de búsqueda de disponibilidad, identificado como el proceso más crítico dentro del flujo funcional del sistema.

La herramienta seleccionada fue Postman, utilizada para simular una carga progresiva de usuarios virtuales concurrentes en una ventana de tiempo controlada. Se configuraron colecciones con múltiples iteraciones y variables dinámicas para registrar las métricas clave del sistema frente a distintos niveles de concurrencia.

Las pruebas se organizaron en tres fases experimentales:

- Escenario base (sin optimización): El sistema monolítico sin optimización.
- Escenario con Elasticsearch: La arquitectura propuesta integrando el motor de indexación distribuida.
- Escenario con Elasticsearch y Redis: El sistema optimizado con la capa de caché distribuido adicional.

Durante cada fase se recopilaron las siguientes métricas de rendimiento:

- Tasa de transferencia (Throughput): Número de solicitudes que el sistema es capaz de procesar por segundo.
- Tiempo promedio de respuesta: Medición del tiempo medio que tarda el sistema en responder a una solicitud.

- Percentiles de latencia (90.º, 95.º, 99.º): Valores que permiten identificar el comportamiento del sistema ante casos extremos.
- Tasa de error: Porcentaje de solicitudes fallidas. Se consideró como error toda respuesta con un código de estado HTTP en los rangos 4xx o 5xx según la semántica definida en el RFC 7231. (Fielding & Reschke, 2014)
    - Errores del cliente (códigos 4xx): por ejemplo, 404 Not Found, 429 Too Many Requests, los cuales indican solicitudes malformadas, recursos inexistentes o límites superados por parte del cliente.
    - Errores del servidor (códigos 5xx): por ejemplo, 500 Internal Server Error, 503 Service Unavailable, que reflejan fallos internos del sistema o incapacidad temporal para atender las solicitudes debido a sobrecarga u otros problemas operativos.

Para una evaluación objetiva, se definieron umbrales de aceptación para cada métrica, tomando como referencia estudios recientes que comparan arquitecturas monolíticas y de microservicios. En particular, se utilizaron los valores empíricos sobre tiempo de respuesta y tasa de error de (Tapia, et al., 2020) y las métricas de consumo de recursos en sistemas escalables de (Blinowski, Ojdowska, & Przybyłek, 2022) La tabla 3 resume los criterios establecidos para validar el desempeño del prototipo.

Métrica	Umbral de Aceptación	Fuente / Justificación
Tiempo promedio de respuesta	≤ 1000 ms	(Tapia, et al., 2020), diferencia crítica entre escenarios monolítico y microservicios
Percentil 95 de latencia	≤ 1500 ms	(Blinowski, Ojdowska, & Przybyłek, 2022), resultados de stress test con JMeter
Tasa de error	≤ 0.5 %	(Tapia, et al., 2020), máximo permitido en microservicios antes de afectar estabilidad
Throughput mínimo esperado	≥ 1.4 solicitudes por segundo bajo carga	(Tapia, et al., 2020), caso optimizado con microservicios
Consumo máximo de CPU	≤ 75 % durante carga sostenida	(Blinowski, Ojdowska, & Przybyłek, 2022), límites aceptables en entorno cloud escalable
Uso máximo de memoria	≤ 80 % de la asignada al contenedor	(Tapia, et al., 2020), pruebas de carga en Docker con Node.js

Tabla 3: Umbrales de aceptación para la validación técnica del prototipo

Con estos escenarios y criterios definidos, se ejecutaron las pruebas de carga. A continuación, se presentan los resultados obtenidos en cada fase, permitiendo medir y discutir el impacto progresivo de cada mejora arquitectónica implementada.

5.1.1 Escenario 1: Sistema monolítico sin optimización

Para establecer una línea base de rendimiento, la configuración monolítica original fue sometida a una prueba de carga. A continuación se detallan los parámetros bajo los cuales se ejecutó la prueba:

Parámetros de la Prueba de Carga (Escenario 1)

- Herramienta Utilizada: Postman Collection Runner
- Endpoint Bajo Prueba: Búsqueda de disponibilidad (findAvailability)
- Nivel de Concurrencia: 100 usuarios virtuales (VUs)
- Duración de la Prueba: 5 minutos
- Solicitudes Totales Generadas: 548

Los resultados obtenidos bajo estas condiciones, resumidos en la Ilustración 24, revelan una degradación severa y un colapso funcional del sistema bajo esta carga. Las métricas clave fueron las siguientes:

- Tiempo de Respuesta Promedio: El sistema tardó una media de 32,140 ms en responder a cada solicitud.
- Tasa de Error: Se registró una tasa de error inaceptable del 68.25%. Como se detalla en la Ilustración 25, la totalidad de los 374 fallos correspondieron al código 502 (Bad Gateway), indicando que el servidor estaba sobrecargado y era incapaz de procesar el flujo de peticiones.
- Capacidad de Procesamiento (Throughput): El rendimiento del sistema se limitó a tan solo 1.78 solicitudes por segundo, lo que confirma la existencia de un cuello de botella crítico.

El análisis de la distribución de la latencia (Ilustración 25) agrava aún más el diagnóstico. Con un percentil 95 º de 39,701 ms, se demuestra que el 5% de los usuarios experimentaron tiempos de espera de casi 40 segundos. En un contexto real, esta latencia es prohibitiva y se traduce directamente en una experiencia de usuario deficiente y en el abandono del servicio. La arquitectura monolítica demuestra ser inviable para soportar la demanda concurrente, fallando en cumplir con los criterios más básicos de rendimiento establecidos para este estudio.

1. Summary

Total requests sent	Throughput	Average response time	Error rate
548	1.78 requests/second	32,140 ms	68.25 %

1.1 Response time

Response time trends during the test duration.

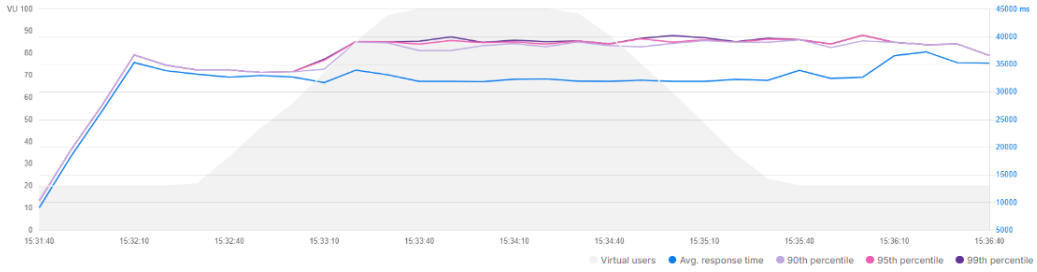

Ilustración 24: Resumen de desempeño del sistema monolítico durante prueba de carga

1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST findAvailability1 https://develop.availability.api.galavail.com/webservice/agencyServices/findAvailability1	32,140	38,682	39,082	39,751	7,608	40,296

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
POST findAvailability1 https://develop.availability.api.galavail.com/webservice/agencyServices/findAvailability1	374	502 Bad Gateway (374)	-	0

2. Metrics for each request

The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST findAvailability1 https://develop.availability.api.galavail.com/webservice/agencyServices/findAvailability1	548	1.78	7,608	32,140	38,682	40,296	68.25

Ilustración 25: Distribución de latencia y errores en el sistema monolítico

5.1.2 Escenario 2: Sistema con indexación distribuida mediante Elasticsearch

En el segundo escenario, se evaluó la arquitectura propuesta con la integración de Elasticsearch como motor de búsqueda, desacoplando así las consultas de disponibilidad del sistema monolítico. La prueba se ejecutó bajo los mismos parámetros de carga para permitir una comparación directa.

Parámetros de la Prueba de Carga (Escenario 2)

- Herramienta Utilizada: Postman Collection Runner
- Endpoint Bajo Prueba: Búsqueda de disponibilidad (findAvailability1)
- Nivel de Concurrencia: 100 usuarios virtuales (VUs)
- Duración de la Prueba: 5 minutos
- Solicitudes Totales Procesadas: 10,768

La introducción de Elasticsearch produjo una mejora de rendimiento drástica y transformacional en todas las métricas, como se observa en la Ilustración 26.

- Capacidad de Procesamiento (Throughput): El rendimiento del sistema se disparó de 1.78 a 34.91 solicitudes por segundo. Esto representa un incremento de casi el 2000% en la capacidad para manejar tráfico concurrente.
- Estabilidad: La tasa de error se desplomó del 68.25% al 0.00%, lo que demuestra que la nueva arquitectura es completamente estable bajo la misma carga que provocó el colapso del monolito.
- Tiempo de Respuesta Promedio: El tiempo de respuesta promedio se redujo de más de 32 segundos a tan solo 217 ms, una mejora de más del 99%.

El análisis detallado de la latencia (Ilustración 27) confirma la solidez de la solución. El percentil 95º se situó en 322 ms, lo que significa que la gran mayoría de los usuarios obtuvieron una respuesta en menos de un tercio de segundo. Es más, incluso en los casos más desfavorables, el percentil 99 º se mantuvo en 540 ms, un valor que sigue siendo órdenes de magnitud inferior al comportamiento promedio

del monolito. Este rendimiento no solo cumple, sino que supera holgadamente todos los criterios de aceptación definidos para el proyecto, demostrando una experiencia de usuario rápida y, sobre todo, predecible.

Estos resultados validan de manera concluyente que la delegación de las operaciones de búsqueda a un motor especializado como Elasticsearch resuelve de forma eficaz los cuellos de botella de rendimiento identificados en la arquitectura monolítica.

1. Summary

Total requests sent	Throughput	Average response time	Error rate
10,768	34.91 requests/second	217 ms	0.00 %

1.1 Response time

Response time trends during the test duration.

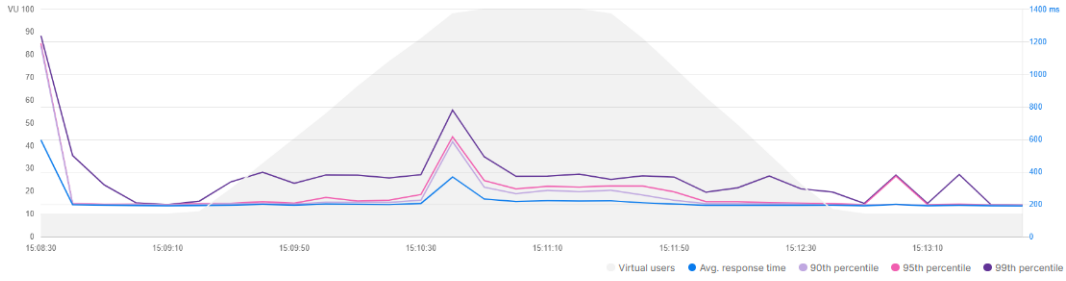

Ilustración 26: Resumen de desempeño del sistema con Elasticsearch durante prueba de carga

1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST findAvailability1 http://localhost:8080/webservice/agencyServices/findAvailability1	217	271	322	520	182	1,240

2. Metrics for each request

The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST findAvailability1 http://localhost:8080/webservice/agencyServices/findAvailability1	10,768	34.91	182	217	271	1,240	0

Ilustración 27: Distribución de latencia y métricas detalladas en el sistema con Elasticsearch

5.1.3 Escenario 3: Sistema con Elasticsearch y Redis

En la fase final, se evaluó la arquitectura completa, añadiendo una capa de caché distribuido con Redis al sistema que ya contaba con Elasticsearch. El objetivo era medir el impacto de la caché en la reducción de la latencia para consultas recurrentes. La prueba se ejecutó bajo los mismos parámetros de carga que los escenarios anteriores.

Parámetros de la Prueba de Carga (Escenario 3)

- Herramienta Utilizada: Postman Collection Runner
- Endpoint Bajo Prueba: Búsqueda de disponibilidad (findAvailability1)
- Nivel de Concurrencia: 100 usuarios virtuales (VUs)
- Duración de la Prueba: 5 minutos
- Solicitudes Totales Procesadas: 12,171

La incorporación de Redis como capa de caché optimizó aún más el rendimiento, logrando tiempos de respuesta de nivel sub-100 milisegundos y una mayor capacidad de procesamiento, como se muestra en la Ilustración 28 e Ilustración 29

- Capacidad de Procesamiento (Throughput): El rendimiento se incrementó nuevamente, alcanzando 39.48 solicitudes por segundo. Esto no solo supera drásticamente al monolito, sino que también representa una mejora del 13% sobre el escenario que solo usaba Elasticsearch, demostrando una mayor eficiencia.
- Estabilidad: La tasa de error se mantuvo en un perfecto 0.00%, confirmando la robustez de la arquitectura final.
- Tiempo de Respuesta Promedio: La métrica más destacada fue la reducción del tiempo de respuesta promedio a tan solo 99 ms. Esto representa una mejora adicional de más del 54% en comparación con el Escenario 2.

El análisis de la latencia demuestra el enorme impacto de la caché. El percentil 95º fue de 97 ms, lo que indica una experiencia de usuario extremadamente rápida y,

sobre todo, consistente. Para ponerlo en perspectiva, incluso el percentil 99º, que representa los casos de respuesta más lenta, se mantuvo en tan solo 285 ms, un umbral de rendimiento que garantiza una fluidez excepcional en la interacción. Esta estabilidad y velocidad son el resultado directo de la eficiencia de Redis: la gran mayoría de las solicitudes se resolvieron casi de forma instantánea al ser servidas desde la memoria, minimizando drásticamente la necesidad de consultar el motor de búsqueda en cada ocasión y optimizando así el uso de los recursos del sistema.

En definitiva, la combinación de Elasticsearch para búsquedas complejas y Redis para el almacenamiento en caché de consultas frecuentes se valida como la configuración óptima, creando un sistema altamente reactivo, estable y capaz de ofrecer una experiencia en tiempo real bajo condiciones de alta concurrencia.

1. Summary

Total requests sent	Throughput	Average response time	Error rate
12,171	39.48 requests/second	99 ms	0.00 %

1.1 Response time

Response time trends during the test duration.

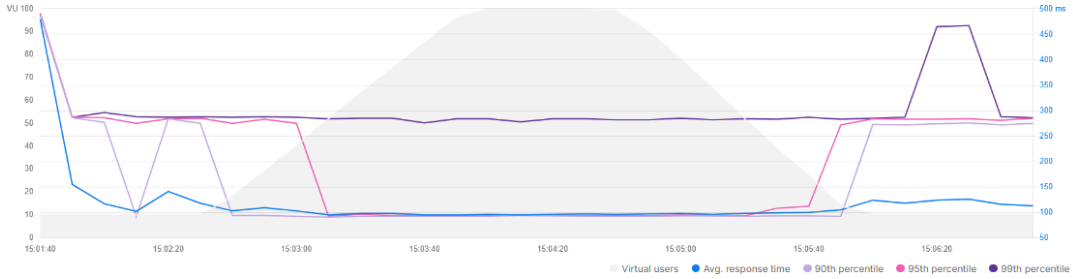

Ilustración 28: Resumen de desempeño del sistema con Elasticsearch y Redis durante prueba de carga

1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST findAvailability1 http://localhost:8080/webservice/agencyServices/findAvailability1	99	93	97	285	89	491

2. Metrics for each request

The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST findAvailability1 http://localhost:8080/webservice/agencyServices/findAvailability1	12,171	39.48	89	99	93	491	0

Ilustración 29: Métricas detalladas de latencia en el sistema con Elasticsearch y Redis

5.2 Evaluación técnica de la arquitectura propuesta basada en microservicios

La sección anterior presentó los resultados cuantitativos de las pruebas de carga, los cuales demostraron mejoras sustanciales en el rendimiento y la estabilidad del sistema tras la implementación de la arquitectura de microservicios (Escenarios 2 y 3) en comparación con la línea base monolítica (Escenario 1). Esta sección profundiza en la evaluación técnica de dicha arquitectura, analizando cómo los componentes tecnológicos clave y las decisiones de diseño e implementación, detalladas en el Capítulo 4, contribuyeron directamente a la consecución de estos resultados y al cumplimiento de los objetivos del proyecto.

El primer componente fundamental en la optimización de las búsquedas fue Elasticsearch, responsable de la indexación dinámica y la recuperación eficiente de los datos de disponibilidad. Como se evidenció en el Escenario 2, su integración resultó en una reducción del tiempo de respuesta promedio de más del 99% y un aumento del throughput de casi el 2000%.

Un aspecto técnico crucial para este desempeño es la correcta definición del esquema de los índices. En la Ilustración 30 se presenta el mapeo de campos utilizado para el índice de disponibilidad, donde la especificación precisa del tipo de datos para cada atributo (ej. keyword para identificadores exactos, date para rangos temporales, long para valores numéricos) es fundamental. Esta configuración permite a Elasticsearch optimizar tanto el almacenamiento como la ejecución de búsquedas, incluyendo consultas complejas, filtros y agregaciones, factor determinante en la drástica reducción de latencia y el aumento de la capacidad de procesamiento observados.

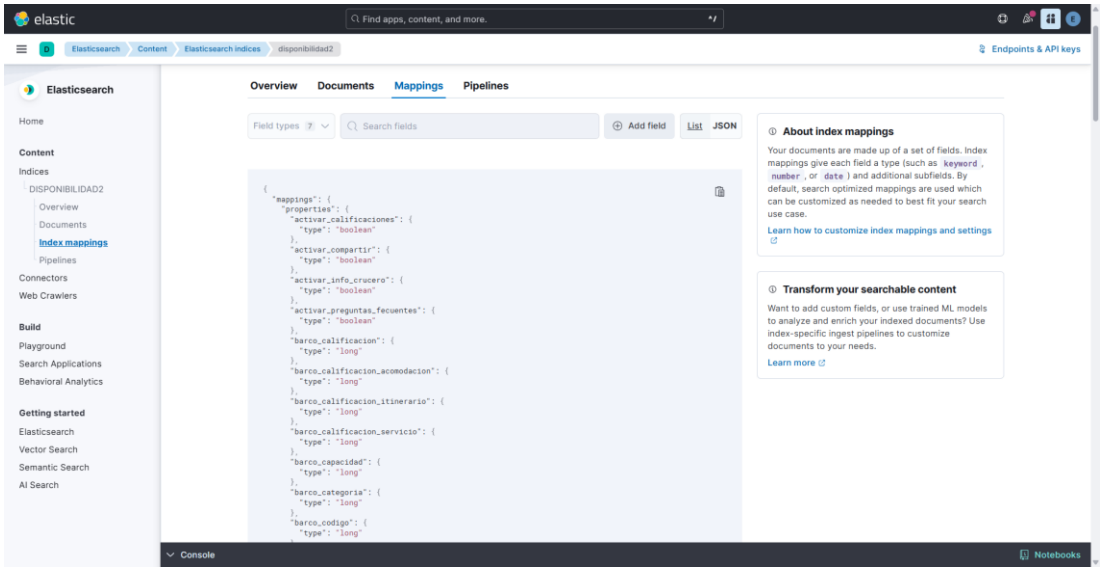

Ilustración 30: Estructura de mapeo de campos en un índice en Elasticsearch

Las capacidades inherentes de Elasticsearch, como su diseño para soportar búsquedas complejas sobre grandes volúmenes de datos, su escalabilidad horizontal y su tolerancia a fallos, fueron fundamentales para los resultados obtenidos. En el contexto del sistema, su implementación no solo facilitó la ejecución de consultas multiparámetro en tiempo real y ordenamientos personalizados por relevancia o prioridad, sino que fue un factor directo en la obtención de una respuesta estable y significativamente más rápida (95º de 322 ms en el Escenario 2), incluso bajo los picos de carga simulados. La transición de una operación de búsqueda secuencial y costosa en el sistema monolítico, a la indexación distribuida y paralelizable de Elasticsearch, fue la transformación clave que permitió la drástica optimización de los tiempos de respuesta y el aumento en la capacidad de procesamiento.

El siguiente componente crucial en la arquitectura es Redis, implementado como capa de caché distribuido. En la Ilustración 31 se observa una representación del estado actual de las claves almacenadas en Redis tras la ejecución del servicio de búsqueda de disponibilidad. Cada clave, como se aprecia, corresponde a una consulta parametrizada que incluye identificadores de agencia, rangos de meses, año y paginación. Esto evidencia el uso de un esquema de almacenamiento

estructurado, diseñado para la recuperación instantánea de resultados previamente solicitados y una gestión granular de la vida útil de la caché.

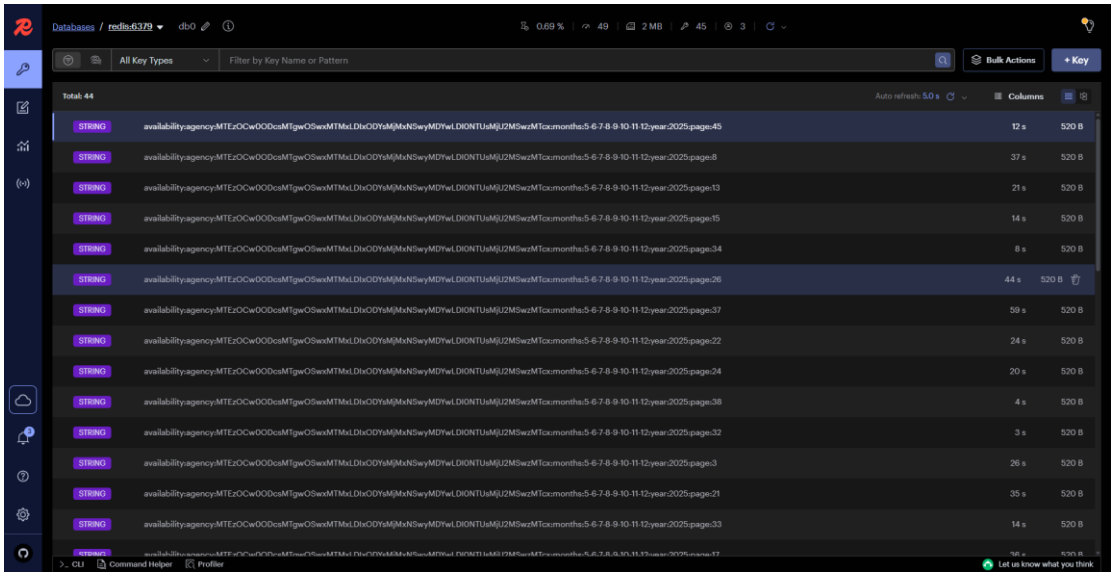

Ilustración 31: Visualización de claves activas y TTL en Redis para consultas de disponibilidad cacheadas

Por su parte, Redis fue incorporado como el mecanismo de almacenamiento en caché para las respuestas de disponibilidad. Esta capa resultó esencial para alcanzar los niveles de rendimiento del Escenario 3, donde el tiempo de respuesta promedio descendió a 99 ms y el 95º a 97 ms. Como se demostró, en contextos de sistemas de reservas con múltiples consultas concurrentes y repetitivas, Redis aporta: una reducción drástica en la latencia al servir datos desde memoria; un alivio de carga significativo sobre Elasticsearch al evitar accesos innecesarios; y, mediante su política de Time-To-Live (TTL), una invalidación controlada que asegura la frescura de los datos cacheados. Su implementación fue, por lo tanto, un factor clave para la optimización final de la latencia y la eficiencia operativa general del sistema.

La sincronización de datos en tiempo real entre el sistema monolítico y los microservicios de la nueva arquitectura se gestionó mediante RabbitMQ. Este bróker de mensajería se configuró para establecer un canal de comunicación asíncrona robusto, por el cual el monolito publica eventos de cambio de disponibilidad (creaciones, modificaciones, cancelaciones). Estos eventos son consumidos por el microservicio de gestión, que actualiza el índice en Elasticsearch y la caché en Redis, aplicando las reglas de negocio necesarias para garantizar la

coherencia de los datos. La Ilustración 32 muestra las métricas operativas del bróker RabbitMQ durante una de las pruebas de carga del Escenario 3. Se observa un flujo constante de publicación y consumo de mensajes, con un número bajo y estable de mensajes en estado 'Ready' (mensajes preparados para ser consumidos, pero aún en cola). Esto indica que los microservicios consumidores procesaron los eventos eficientemente, sin acumulación de colas, lo cual es crucial para mantener la consistencia eventual de los datos con una latencia mínima en la propagación de cambios.

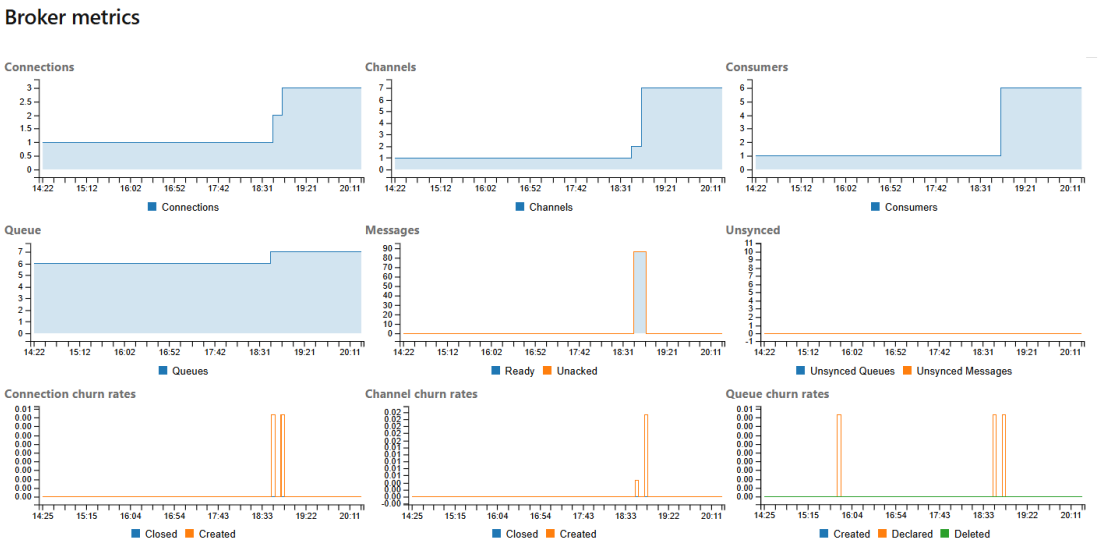

Ilustración 32: Métricas de operación del broker RabbitMQ durante pruebas de carga

Este mecanismo de comunicación asíncrona, cuya correcta operación se evidencia en las métricas del bróker, fue esencial para: asegurar la consistencia eventual de los datos en un sistema distribuido con alta disponibilidad; reducir la dependencia directa y los acoplamientos síncronos entre el monolito y los nuevos servicios; y permitir el manejo de picos de actualizaciones sin congestionar la base de datos principal del sistema legado.

Finalmente, se incorporó un API Gateway como punto centralizado de acceso a los microservicios públicos, lo que facilitó la transición gradual desde el sistema monolítico hacia una arquitectura distribuida. Gracias a esta capa, fue posible exponer de manera independiente los servicios encargados de la búsqueda de disponibilidad, manteniendo el resto del sistema bajo su lógica tradicional. El uso

del API Gateway permitió independizar y versionar endpoints sin afectar el sistema heredado, simplificar la gestión de acceso, seguridad y escalado, así como reforzar la interoperabilidad con sistemas externos.

La Ilustración 33 presenta las métricas clave de rendimiento observadas en el API Gateway durante las pruebas de carga. Esta evidencia del comportamiento estable y eficiente del Gateway bajo carga demuestra su capacidad para gestionar el flujo de solicitudes sin introducir cuellos de botella significativos, lo cual fue esencial para el rendimiento global del sistema optimizado. Adicionalmente, su configuración mediante archivos versionables demostró facilitar una gestión ágil de las rutas, validando el principio de "extensión sin disrupción" en la práctica.

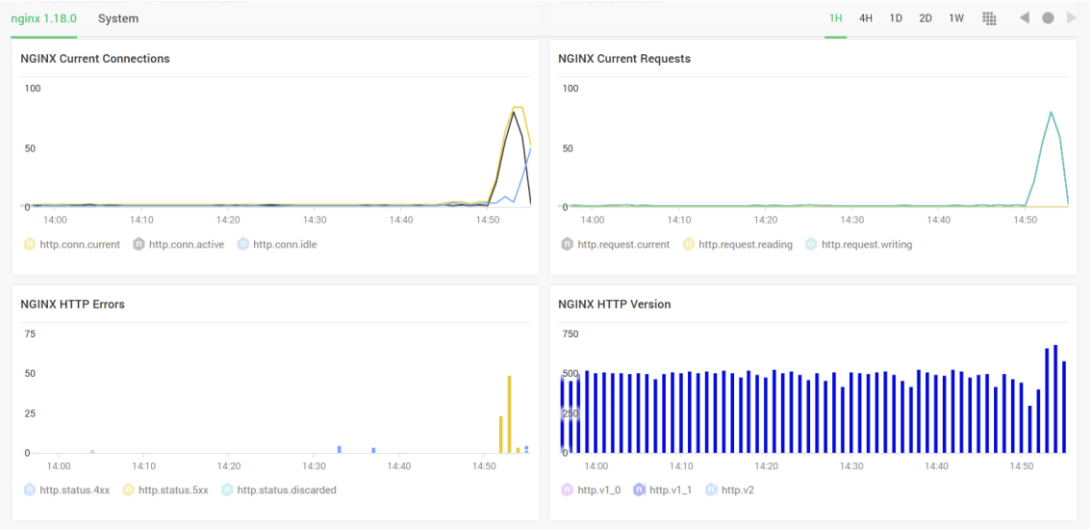

Ilustración 33: Métricas Clave de Rendimiento del API Gateway (Nginx) durante Pruebas de Carga

Para culminar la evaluación técnica de la arquitectura implementada y ofrecer una perspectiva integral de sus ventajas, la Tabla 4 proporciona una comparativa exhaustiva. Esta tabla resume y contrasta directamente el comportamiento y las características del sistema monolítico original frente a la solución basada en microservicios desarrollada en este proyecto. Se analizan atributos técnicos fundamentales, destacando las mejoras significativas logradas en aspectos cruciales como el rendimiento bajo carga, la escalabilidad horizontal, la modularidad del sistema y la mantenibilidad general, validando así la efectividad de la transformación arquitectónica.

Criterio	Sin componentes (Monolítico)	Con componentes (Microservicios + ES + Redis + RabbitMQ)
Tiempo promedio de respuesta	> 30 s bajo carga alta	< 100 ms promedio
Manejo de concurrencia	Limitado	Alta, validada en pruebas
Escalabilidad	Vertical únicamente	Horizontal, validada técnicamente
Disponibilidad de datos en tiempo real	Limitada, dependiente de DB central	Sí, mediante indexación distribuida
Flexibilidad para nuevos servicios	Requiere cambios en toda la aplicación	Alta, gracias a diseño desacoplado
Carga sobre la base de datos principal	Alta	Reducción significativa vía Redis
Modularidad del sistema	Baja	Alta
Consistencia de datos entre componentes	Dependiente de integraciones sincrónicas	Consistencia eventual mediante RabbitMQ
Uso de caché	No implementado	Implementado con TTL dinámico
Tolerancia a fallos	Baja	Alta

Tabla 4: Comparativa técnica entre arquitectura monolítica y arquitectura propuesta basada en microservicios

En conjunto, la arquitectura propuesta demostró ser técnicamente viable, eficiente y adaptable a las necesidades de escalabilidad y respuesta en tiempo real que exige un sistema de reservas empresarial moderno. Su diseño modular, la utilización de tecnologías especializadas y el enfoque en la optimización de la consulta de disponibilidad constituyen aportes concretos para entornos que requieren modernización sin incurrir en la reescritura total del software legado.

5.3 Impacto de la indexación dinámica y distribuida sobre la experiencia del usuario

La incorporación de mecanismos de indexación dinámica y distribuida dentro de la arquitectura propuesta ha generado un impacto directo y positivo sobre la experiencia del usuario final. En el contexto de sistemas empresariales de reservas, donde los tiempos de respuesta influyen de manera crítica en la satisfacción del cliente, la capacidad de entregar resultados en tiempo real constituye una ventaja competitiva significativa. Esta mejora se materializa a través de la reducción de la latencia visible, la estabilidad del sistema ante picos de tráfico y la entrega

consistente de resultados incluso bajo condiciones de alta concurrencia (Horváth, Sakhnenko, & Gurbál, 2024) & (He, 2020).

Uno de los beneficios más relevantes percibidos a nivel de usuario es la reducción del tiempo necesario para obtener respuestas a consultas complejas. Mientras que en la arquitectura monolítica original los tiempos de espera podían alcanzar los 32 segundos en promedio (Escenario 1), la implementación de Elasticsearch permitió realizar búsquedas distribuidas con una respuesta promedio de 217 ms (Escenario 2), transformando radicalmente la experiencia de navegación, haciéndola más fluida y eficiente, especialmente en escenarios donde múltiples parámetros de búsqueda deben ser evaluados simultáneamente (Mishra, Jaiswal, Prakash, & Barwal, 2022).

Además de la velocidad, la consistencia en los resultados fue fortalecida gracias al uso de Redis como capa de almacenamiento temporal. Este componente permite atender consultas repetidas de manera instantánea, evitando recalcular información previamente obtenida. Desde el punto de vista del usuario, esto se traduce en una interfaz más ágil y sin interrupciones, incluso cuando múltiples usuarios acceden al sistema en paralelo. La combinación de indexación distribuida con estrategias de caché adecuadas no solo mejora la capacidad de respuesta, sino que garantiza una experiencia homogénea en diferentes momentos del día, independientemente de la carga del sistema (Rahmatulloh, Nugraha, Gunawan, & Darmawan, 2022).

Otro aspecto relevante es la predictibilidad del sistema ante escenarios de carga elevada. Las pruebas realizadas demostraron que, gracias a la arquitectura modular y la distribución eficiente de tareas, el tiempo de respuesta se mantuvo estable a lo largo de toda la ejecución, sin variaciones bruscas. Esta estabilidad percibida por el usuario es clave para generar confianza en plataformas de reservas, donde cada segundo puede determinar la concreción o el abandono de una operación (Horváth, Sakhnenko, & Gurbál, 2024).

La arquitectura también permite una experiencia más robusta durante procesos de búsqueda masiva o simultánea. En sistemas tradicionales, este tipo de operaciones solía causar bloqueos o ralentizaciones generalizadas. Sin embargo, en el nuevo modelo, la separación de responsabilidades entre servicios y el uso de tecnologías orientadas al procesamiento paralelo aseguran que cada solicitud sea tratada de manera independiente, preservando la calidad del servicio ofrecido al usuario final (He, 2020) & (Mishra, Jaiswal, Prakash, & Barwal, 2022).

En conjunto, los resultados obtenidos validan que la aplicación de indexación dinámica y almacenamiento distribuido no solo responde a requerimientos técnicos, sino que también incide de manera directa en la percepción de calidad del sistema. Esta mejora integral en la experiencia del usuario es consecuencia del rediseño arquitectónico orientado al rendimiento, la resiliencia y la adaptabilidad, pilares fundamentales en el desarrollo de soluciones tecnológicas modernas para entornos empresariales exigentes.

6. Conclusiones

La presente investigación aplicada ha demostrado que la implementación de una arquitectura basada en microservicios con indexación dinámica y distribuida constituye una solución técnicamente viable y efectiva para resolver problemas críticos de rendimiento y escalabilidad en sistemas empresariales que operan bajo arquitecturas monolíticas. Los resultados experimentales obtenidos permiten concluir lo siguiente:

1. La migración gradual hacia microservicios mediante el patrón Strangler ha demostrado ser una estrategia altamente efectiva para modernizar sistemas legados sin interrumpir su operatividad. Este enfoque permitió realizar una transición ordenada y controlada, reduciendo significativamente los riesgos operativos asociados a los procesos de transformación arquitectónica.
2. El uso de tecnologías especializadas como Elasticsearch para indexación distribuida y Redis para almacenamiento en caché ha generado mejoras sustanciales en la capacidad de respuesta del sistema. Las pruebas realizadas evidencian reducciones drásticas en los tiempos de respuesta promedio (de más de 30 segundos a menos de 100 milisegundos), lo cual repercute directamente en la experiencia del usuario y en la eficiencia operativa del negocio. Este resultado confirma la importancia estratégica de seleccionar herramientas tecnológicas que respondan específicamente a los requerimientos funcionales y operativos del sistema.
3. La integración de RabbitMQ como mecanismo de sincronización asincrónica entre la arquitectura monolítica y los microservicios demostró ser efectiva para garantizar coherencia y consistencia eventual en entornos distribuidos. Este enfoque permitió establecer una comunicación desacoplada y confiable, facilitando la incorporación progresiva de servicios independientes sin comprometer el rendimiento del sistema ni saturar la base de datos central.
4. La implementación del API Gateway ha simplificado la gestión, exposición y seguridad de los microservicios, favoreciendo la independencia funcional y técnica de cada componente. Su incorporación confirmó la importancia de

contar con una capa intermedia que gestione el tráfico y la interoperabilidad en entornos híbridos, facilitando así la coexistencia de diferentes paradigmas tecnológicos.

5. La arquitectura propuesta también validó la efectividad del escalamiento horizontal en entornos basados en contenedores. La utilización de Docker y Docker Compose permitió escalar servicios específicos según la demanda, optimizando el uso de recursos y facilitando la gestión operativa del sistema. Esto ha demostrado ser una ventaja competitiva clave en la gestión de plataformas digitales que requieren alta disponibilidad y rendimiento predecible.
6. Las pruebas de carga con 100 usuarios virtuales validaron empíricamente la robustez de la arquitectura propuesta, el tiempo de respuesta promedio se redujo de más de 30 segundos en el monolito a solo 99 ms en la solución final con Elasticsearch y Redis, mientras la tasa de error descendió del 68.25% al 0%. Estos resultados confirman que la arquitectura de microservicios implementada ofrece una respuesta significativamente más eficiente, estable y escalable, posicionándola como una alternativa superior frente al enfoque monolítico.
7. Finalmente, esta investigación resalta la relevancia del estudio avanzado en diseño de arquitecturas de sistemas llevado a cabo en la presente Maestría en Software. La aplicación práctica de los conocimientos adquiridos en torno a microservicios (Patrones de Diseño de APIs), tecnologías avanzadas relacionadas con sistemas distribuidos (Sistemas Distribuidos Avanzados y Patrones de Diseño de Software), así como la implementación de estrategias modernas de integración (Patrones de Integración Empresarial) y despliegue continuo (Automatización de los Procesos de Software), evidencia claramente el valor del perfil profesional orientado a la innovación tecnológica y a la gestión estratégica del software en escenarios empresariales actuales y exigentes.

En conclusión, el desarrollo de este proyecto ha generado aportes significativos al conocimiento práctico y teórico sobre la migración e implementación de arquitecturas modernas en entornos empresariales, demostrando que las metodologías ágiles y las tecnologías especializadas no solo resuelven desafíos

técnicos, sino que también impulsan la evolución estratégica y competitiva de las organizaciones que deciden adoptarlas. En un contexto marcado por el darwinismo digital, donde solo aquellas entidades capaces de adaptarse tecnológicamente logran mantenerse relevantes, la transformación arquitectónica del software se posiciona no solo como una ventaja, sino como una condición indispensable para la supervivencia organizacional en la era digital.

Referencias

Barua, B., & Kaiser, M. (2024). *Novel architecture for distributed travel data integration and service provision using microservices.* Dhaka, Bangladesh: BGMEA University of Fashion & Technology & Jahangirnagar University.

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access, 10*, 20357–20373.

Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *The Journal of Systems and Software*, 77–97.

Fielding, R., & Reschke, J. (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* Internet Engineering Task Force (IETF). RFC Editor. Obtenido de Mozilla: https://datatracker.ietf.org/doc/html/rfc7231

Hasan, M. H., Osman, M. H., Admodisastro, N. I., & Muhammad, M. S. (2023). Legacy systems to cloud migration: A review from the architectural perspective. *The Journal of Systems and Software, 35*(4).

He, J. (2020). Research on Personalized Search Based on ElasticSearch. *Proceedings of the 2nd International Conference on Civil Aviation Safety and Information Technology (ICCASIT)* (págs. 572–575). Chengdu, China: IEEE.

Horváth, M., Sakhnenko, V., & Gurbál, F. (2024). Comparison of scalability and performance in microservices and monolithic architectures. *Proceedings of the 2024 IEEE 17th International Scientific Conference on Informatics* (págs. 82-87). Košice, Eslovaquia: IEEE.

Laigner, R., Zhang, Z., Liu, Y., Gomes, L. F., & Zhou, Y. (2025). Online marketplace: A benchmark for data management in microservices. *Proceedings of the ACM on Management of Data (SIGMOD). 29.* Washington, DC, USA: Association for Computing Machinery.

Li, C.-Y., Ma, S.-P., & Lu, T.-W. (2020). Microservice migration using Strangler Fig pattern: A case study on the Green Button system. *2020 International Computer Symposium (ICS).* Keelung, Taiwan: IEEE.

Mishra, R., Jaiswal, N., Prakash, R., & Barwal, P. N. (2022). Transition from monolithic to microservices architecture: Need and proposed pipeline. *Proceedings of the 2022 International Conference on Futuristic Technologies (INCOFT)* (págs. 1-6). Karnataka, India: IEEE.

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (Segunda edición ed.). Sebastopol, CA: O'Reilly Media.

Rahmatulloh, A., Nugraha, F., Gunawan, R., & Darmawan, I. (2022). Event-driven architecture to improve performance and scalability in microservices-based systems. *Proceedings of the 2022 International Conference on Advancement in Data Science, E-learning and Information Systems (ICADEIS)* (págs. 1–6). Indonesia: IEEE.

Stojanova, Z., Hristoski, I., Stojanova, J., & Stojkova, A. (2023). A tertiary study on microservices: Research trends and recommendations. *Programming and Computer Software*, 796-821.

Tapia, F., Mora, M., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance. *Applied Sciences*.

Wang, Z., Yu, Z., & Tang, F. (2020). Design and implementation of a health status reporting system based on Spring Boot. *Proceedings of the 2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)* (págs. 453-457). Beijing, China: IEEE.