Facultad de Ciencia y Tecnología (FCyT)

Sede Oro Verde

Tesina

Carrera: Licenciatura en Sistemas de Información

Tema:

Proponer una metodología para rediseñar un sistema ERP legacy monolítico hacia una arquitectura de microservicios.

Autor

Sergio Ricardo Montero

Director de Tesis

Ing. Emanuel Goette

Co-Director de Tesis

Lic. Roberto Antola

Revisores

Ing. Mónica Vera

Lic. Pamela Bonadeo

Lic. Ignacio Cabrera

Año 2025

Dedicatoria: dedico este trabajo a mi esposa Ximena por sus consejos, su apoyo incondicional y su paciencia a lo largo de todo este proceso. También a mis hijos, Luciano y Gerónimo. Un fuerte y sentido abrazo para todos ellos.

Agradecimientos: agradezco a mis padres, que me guiaron en mis comienzos, a mi director Emanuel Goette y a mi co-director Roberto Antola, que me ayudaron a que esta tesina fuera posible y a todas aquellas personas que directa e indirectamente han contribuido de una u otra forma a mi desarrollo profesional y humano. A todos ellos, muchas gracias.

ÍNDICE GENERAL

Índice de imágenes ____ ***6***

Resumen ____ ***7***

Introducción ____ ***8***

Objetivos ____ ***8***

Objetivo general ____ **8**

Objetivos secundarios ____ **8**

Capítulo I – Fundamentos conceptuales y arquitectónicos ____ ***9***

1.1 Sistemas Legacy ____ **9**

1.2 Sistemas ERP ____ **10**

1.2.1 ¿Qué es un ERP? ____ 10

1.2.2 Evolución histórica ____ 11

1.3 Arquitectura monolítica ____ **12**

1.3.1 Sistemas monolíticos ____ 12

1.3.2 Ventajas ____ 13

1.3.3 Desventajas ____ 13

1.4 Arquitectura de microservicios ____ **14**

1.4.1 Microservicios ____ 14

1.4.2 Características de la arquitectura de microservicios ____ 16

1.4.3 Ventajas ____ 17

1.4.4 Desventajas ____ 18

Capítulo II – Tecnologías y patrones de diseño ____ ***20***

2.1 Stack tecnológico ____ **20**

2.2 Contenedores ____ **21**

2.3 Orquestación de contenedores ____ **22**

2.4 Integración continua (CI) y Distribución continua (CD) ____ **23**

2.5 Computación en la nube ____ **24**

2.5.1 Tipos de nube ____ 24

2.5.2 Servicios de la nube ____ 24

2.5.3 Beneficios de la computación en la nube para las arquitecturas de microservicios ____ 25

2.6 Patrones de diseño ____ **26**

2.6.1 Patrón Estrangulador (*Strangler Fig*) ____ 27

2.6.2 Patrón Rama por Abstracción (*Branch By Abstraction*) ____ 28

2.6.3 Patrón Ejecución Paralela (Paralell Run) ____ 34

2.6.4 Patrón Puerta de Enlace (Api Gateway) ____ 35

2.6.5 Patrón Reintentar (*Retry*) ____ 36

2.6.6 Patrón Corta Circuitos (*Circuit Breaker*) ____ 37

Capítulo III – Principios de diseño y dinámica estructural de los microservicios ____ ***39***

3.1 Comunicación en una arquitectura de microservicios 39
3.1.1 Comunicación sincrónica 39
3.1.2 Comunicación asincrónica 40
3.1.3 Tipos de comunicación 40
3.2 Transacciones 41
3.2.1 Propiedades ACID 41
3.2.2 Transacciones distribuidas 42
3.2.3 Sagas 42
3.3 Cohesión 44
3.3.1 Niveles de cohesión 44
3.4 Acoplamiento 45
3.4.1 Tipos de acoplamiento 45
3.5 Relación entre acoplamiento y cohesión 46
Capítulo IV – Estrategias y enfoques para la transición del monolítico a microservicios 47
4.1 Diseño dirigido por el dominio (DDD) 47
4.2 Enfoques para formular microservicios a partir de un sistema monolítico 48
4.2.1 Enfoque del Big Bang 48
4.2.2 Enfoque incremental Strangler Fig 49
4.2.3 Enfoque propuesto por Richarson y Smith 50
4.3 Pruebas de conceptos (PoC) 55
4.3.1 Definición de una PoC 55
4.3.2 Aplicación de las PoC en el desarrollo del software 55
4.3.3 Importancia de la PoC 55
4.3.4 Componentes clave de la PoC 55
4.3.5 Beneficios de crear una PoC 56
4.4 Equipos interdisciplinarios 56
Capítulo V – Metodología propuesta para el rediseño de un ERP legacy monolítico hacia microservicios 58
5.1 Definición conceptual de la metodología 58
5.2 Etapa 1 - Elaboración del diagnóstico 59
5.2.1 Formación un equipo de trabajo interdisciplinario 60
5.2.2 Entregables del diagnóstico 60
5.3 Etapa 2 - Prueba de conceptos (PoC) 62
5.3.1 Fase 1 – Validación de la infraestructura tecnológica 62
5.3.2 Fase 2 – Validación de mecanismos transversales 62
5.3.3 Fase 3 – Validación del enfoque Strangler Fig 64
5.3.4 Fase 4 – Validación organizacional 65
5.4 Etapa 3 – Diseño incremental 65
5.4.1 Criterios de extracción de módulos 65
5.4.2 Proceso iterativo de diseño incremental 66
Conclusión 68

Futuras investigaciones _____ *69*

Bibliografia _____ *70*

Índice de imágenes

Figura 1 – Módulos principales de un ERP 11
Figura 2 – Evolución histórica del ERP 12
Figura 3 – Microservicio exponiendo su funcionalidad a través de una API REST y un tópico 16
Figura 4 – Patrón estrangulador 28
Figura 5 – Patrón Rama por Abstracción - Paso 1 30
Figura 6 – Patrón Rama por Abstracción - Paso 2 30
Figura 7 – Patrón Rama por Abstracción - Paso 3 31
Figura 8 – Patrón Rama por Abstracción - Paso 4 32
Figura 9 – Patrón Rama por Abstracción - Paso 5 33
Figura 10 – Patrón Ejecución Paralela - Ejemplo de una ejecución paralela 35
Figura 11 – Nueva funcionalidad implementada como servicio 51
Figura 12 – Refactorizando una aplicación existente 52
Figura 13 – Refactorizando el módulo Z del monolítico en microservicios 54
Figura 14 – Metodología propuesta para el rediseño de un sistema ERP legacy monolítico a microservicios 59

Resumen

Durante décadas, las arquitecturas monolíticas fueron ampliamente utilizadas en el desarrollo de sistemas, en un contexto donde no existía una necesidad constante de cambios ni se requerían sistemas flexibles o altamente escalables. Sin embargo, la creciente demanda de requerimientos, la necesidad de actualizaciones frecuentes, la inmediatez en la implementación de cambios y el aumento de la performance según la demanda representan desafíos significativos en este tipo de arquitectura. En contraste, los sistemas basados en microservicios ofrecen ventajas sustanciales en términos de desarrollo de nuevas funcionalidades, escalabilidad y despliegue continuo. Este enfoque arquitectónico ha ganado popularidad en los últimos años dentro de la comunidad de desarrollo de software y se presenta como una alternativa prometedora. La presente tesina tiene como objetivo principal proponer una metodología que permita abordar el proceso de rediseño de un sistema ERP[1] *legacy*[2] monolítico[3] hacia una arquitectura basada en microservicios, proporcionando fundamentos teóricos, lineamientos generales y un orden estructurado de las actividades necesarias.

Palabras claves: monolíticos, microservicios, *legacy*, ERP.

[1] Las siglas ERP, en inglés “Entreprise resource planing”, que se traduce como “Planificación de recursos empresariales”, es lo que se conoce como sistemas ERP.

[2] Los sistemas legacy, también llamados heredados o legados, se refieren a sistemas informáticos

[2] Los sistemas legacy, también llamados heredados o legados, se refieren a sistemas informáticos desarrollados décadas anteriores a la actual.

[3] Para Lewis & Fowler (2014), un monolítico es una aplicación construida como una sola unidad.

Introducción

El rediseño de un sistema mediante el cambio de su enfoque arquitectónico representa un desafío significativo. Esta tesina se centra en el estudio del rediseño de un sistema ERP *legacy*, desarrollado bajo una arquitectura monolítica, hacia una arquitectura basada en microservicios. El propósito principal del trabajo es proponer una metodología que, sustentada en bases teóricas y lineamientos generales, facilite dicho proceso de transformación.

Es importante destacar que la metodología propuesta se plantea como un marco estructural de trabajo que integra diversos enfoques con el objetivo de guiar y simplificar la migración desde un modelo monolítico hacia uno orientado a microservicios.

De acuerdo con sus objetivos extrínsecos, esta investigación se clasifica como aplicada. El alcance del trabajo se limita al diseño de las etapas metodológicas, sin contemplar la implementación práctica en un caso real.

Objetivos

Objetivo general

Proponer una metodología de trabajo para abordar el rediseño de un sistema ERP *legacy*, de arquitectura monolítica en uno de arquitectura basada en microservicios.

Objetivos secundarios

- Investigar sobre sistemas ERP, arquitecturas monolíticas, arquitecturas basadas en microservicios, patrones de diseño y computación en la nube.
- Establecer criterios para formular los microservicios a partir del monolítico.
- Analizar los patrones de diseño de microservicios, que contribuyan con el rediseño del sistema.
- Diseñar las etapas de la metodología de trabajo para abordar el rediseño de un sistema ERP *legacy*, de arquitectura monolítica en uno de arquitectura basada en microservicios.

Capítulo I – Fundamentos conceptuales y arquitectónicos

En este capítulo se presentan los conceptos fundamentales relacionados con los sistemas *legacy*, los sistemas ERP, las arquitecturas monolíticas y las arquitecturas basadas en microservicios. Se realiza un recorrido histórico sobre la evolución de los sistemas ERP, desde sus orígenes hasta su desarrollo en la actualidad. Finalmente, se analizan las principales ventajas y desventajas de las arquitecturas monolíticas y de microservicios.

1.1 Sistemas Legacy

Los sistemas *legacy* o sistemas legados, según Genexus (2025)[4], son soluciones de software antiguas y desactualizadas que continúan en funcionamiento debido a que aún satisfacen las necesidades para las cuales fueron originalmente desarrolladas.

Ulrich (1994),[5] los describe como sistemas independientes construidos en una era tecnológica anterior, que carecen de una documentación adecuada y cuya estructura dificulta su comprensión o modificación. En esta misma línea, Bennett (1995),[6] los define de manera informal como grandes sistemas de software cuya complejidad supera las capacidades técnicas actuales de los equipos de desarrollo; sin embargo, continúan siendo esenciales para el funcionamiento de la organización.

En términos generales, los sistemas *legacy* suelen estar vinculados a plataformas tecnológicas obsoletas, lenguajes de programación en desuso, aplicaciones con arquitecturas monolíticas o procesos operativos que ya no se ajustan fácilmente a los contextos tecnológicos contemporáneos. Aunque desempeñan un rol crítico dentro de las operaciones de una organización, suelen presentar importantes desafíos en cuanto a su mantenimiento, escalabilidad y modernización.

Según Whitestack (2025)[7], estos sistemas se caracterizan por su ineficiencia y la dificultad de ser adaptados o mantenidos en entornos donde la tecnología avanza de forma constante. Por lo general, se encuentran alojados en infraestructuras locales, lo que impide la escalabilidad y flexibilidad que demandan las empresas actuales. Esto

[4] Genexus es una empresa de desarrollo de software originaria de Uruguay, reconocida por su plataforma homónima de desarrollo low-code, utilizada para generar aplicaciones empresariales de forma automatizada. Su experiencia en modernización de sistemas y desarrollo ágil la posiciona como una fuente relevante en el ámbito tecnológico latinoamericano.

[5] Ulrich William es un reconocido consultor y autor especializado en estrategias de modernización de sistemas de información. Su obra *Legacy Systems: Transformation Strategies* (1994) es una referencia clave en procesos de transformación de sistemas heredados hacia arquitecturas modernas.

[6] Bennett Kevin fue un académico británico vinculado a la Universidad de Durham, experto en evolución y mantenimiento de *software*. Es ampliamente citado por sus contribuciones al estudio de sistemas *legacy* y la ingeniería de software a gran escala.

[7] Whitestack es una empresa tecnológica especializada en soluciones de infraestructura abierta y transformación digital. Su trabajo se centra en tecnologías como cloud computing, redes definidas por software (SDN) y modernización de sistemas legacy mediante arquitecturas abiertas y escalables.

conlleva altos costos operativos, restricciones en el crecimiento del sistema, y una creciente exposición a riesgos de seguridad.

1.2 Sistemas ERP

1.2.1 ¿Qué es un ERP?

Las siglas ERP, en inglés "*Entreprise resource planing*", que se traduce como "Planificación de recursos empresariales", es lo que se conoce como sistemas ERP.

Esteves y Pastor (1999), lo definen como paquetes de software, compuesto por varios módulos, tales como, recursos humanos, ventas, finanzas y producción, que posibilitan la integración de datos a través de procesos de negocios. Estos paquetes de software permiten ser configurados para responder a las necesidades específicas de cada organización.

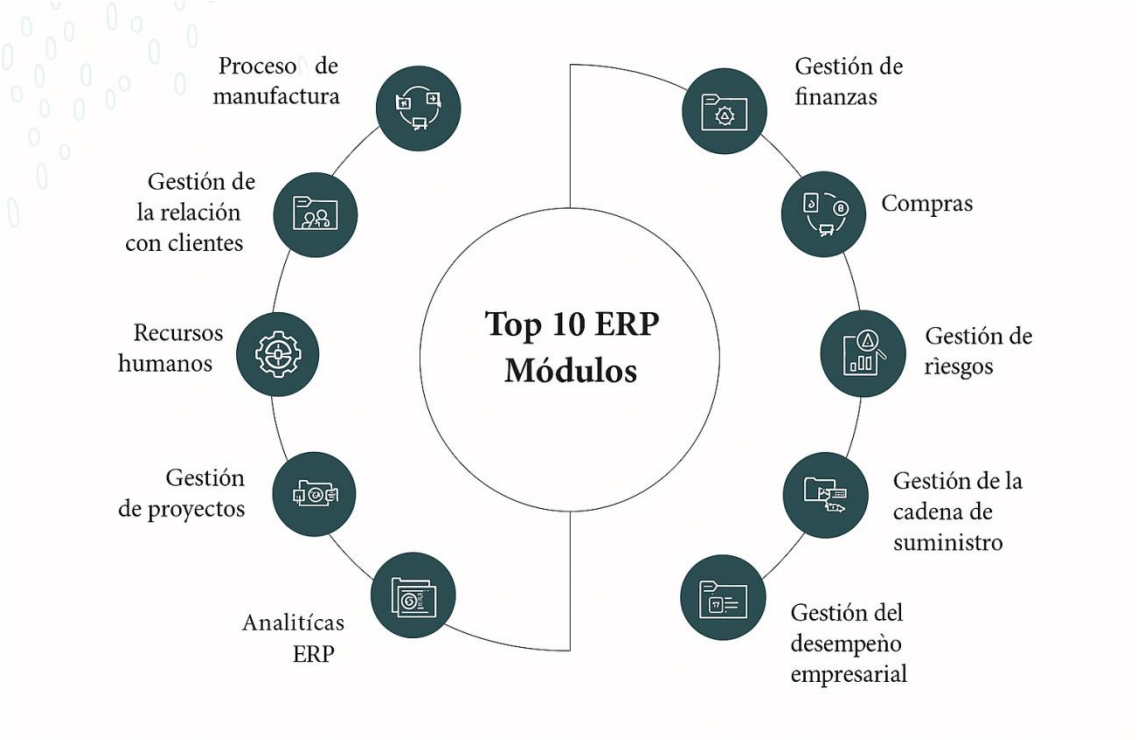
Oracle[8] (2024), los ERP, enlazan una multitud de procesos empresariales, facilitando el flujo de datos entre ellos, permitiendo automatización e inteligencia esencial, para llevar adelante de manera eficiente las operaciones del negocio. En la actualidad son considerados elementos imprescindibles para la gestión de miles de empresas de tamaños e industrias diferentes.

Este tipo de sistema es utilizado por las organizaciones para gestionar las actividades diarias, como las ventas, compras, fabricación, operaciones de la cadena de suministro, finanzas, contabilidad, aprovisionamiento, gestión de proyectos, gestión de relaciones con el cliente (CRM), gestión del capital humano, entre otros. Los ERP suelen integrarse con otros sistemas como plataformas de comercio electrónico e incluso otros ERP. En la Figura 1 se muestra el ERP de Oracle con sus diez módulos principales.

[8] Oracle es una empresa tecnológica multinacional estadounidense especializada en software empresarial, bases de datos y servicios en la nube. Fue fundada en 1977 por Larry Ellison, Bob Miner y Ed Oates.

Figura 1

Módulos principales de un ERP



Nota. Adaptada de Oracle (2024) https://www.oracle.com/ar/erp/erp-modules/

1.2.2 Evolución histórica

Oracle (2024), el ingeniero Ford Whitman Harris[9] desarrolló en 1913, lo que posteriormente se conoció como el modelo de cantidad económica (EOQ), un sistema de fabricación basado en papel para la programación de la producción. Este modelo fue un estándar durante décadas. En 1964, el fabricante de herramientas Black and Decker adoptó una solución de planificación de requerimiento de materiales (MRP), que combinaba los conceptos del modelo EOQ con una computadora *mainframe*. El MRP fue el estándar de fabricación, hasta que se desarrolló la planificación de recursos de fabricación (MRP II) en 1983. La MRP II, presentaba los “módulos” como componente arquitectónico clave del software y los componentes de fabricación básicos integrados, incluían las compras, las listas de materiales, la programación y la gestión de contratos. La MRP II, integraba por primera vez diferentes tareas de fabricación en un sistema común. Con el avance tecnológico de la informática se desarrollaron durante los años setenta y ochenta conceptos similares a la MRP II, para abordar actividades comerciales distintas de la fabricación y se incorporaron las finanzas, la gestión de las relaciones con

[9] Ford Whitman Harris (1877-1962) fue un ingeniero de producción estadounidense que derivó la fórmula de raíz cuadrada para ordenar el inventario, ahora conocida como cantidad económica de pedido.

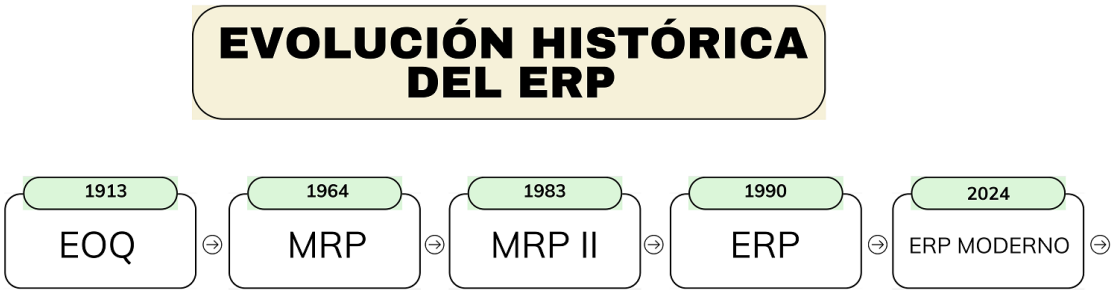
los clientes y datos de recursos humanos. En el año 1990, los analistas tecnológicos encontraron un nombre para esta nueva categoría de software de gestión empresarial: la planificación de recursos empresariales (ERP).

Con el paso del tiempo, continuaron evolucionando para adaptarse a las cambiantes necesidades empresariales y tecnológicas. Se han agregado características como analítica avanzada, computación en la nube e inteligencia artificial para mejorar la funcionalidad y la usabilidad de los sistemas ERP.

Esta evolución ha sido impulsada por los avances en la tecnología de la información, la transformación de los modelos de negocio y la necesidad de las organizaciones de adaptarse a un entorno empresarial cada vez más globalizado y competitivo. En la Figura 2, se observa la evolución del ERP, conforme pasaron los años.

Figura 2

Evolución histórica del ERP



1.3 Arquitectura monolítica

La arquitectura monolítica es un estilo de diseño de software en el que una aplicación es construida como una sola unidad, donde todos sus módulos están interconectados y desplegados como una única entidad.

1.3.1 Sistemas monolíticos

Para Lewis[10] & Fowler[11] (2014), un monolítico es una aplicación construida como una sola unidad; normalmente dividida en tres partes principales: una interfaz de usuario del lado del cliente (que consta de páginas HTML y JavaScript, que se ejecutan en el navegador del cliente), una base de datos, generalmente relacional, administrada por un sistema de gestión de bases de datos y una aplicación del lado del servidor que maneja

[10] Lewis es un experto reconocido internacionalmente en arquitectura y diseño de software. Es arquitecto de software y director de Thoughtworks.

[11] Fowler (1963), es un desarrollador de software británico, autor y orador público internacional sobre desarrollo de software. Ha escrito numerosos libros y artículos.

las peticiones HTTP, ejecuta la lógica del dominio y se encarga de recuperar y actualizar la información en la base de datos. La aplicación servidor, constituye un único ejecutable lógico. Cualquier cambio en el sistema, implica crear una nueva versión de toda la aplicación servidor.

Para Richardson & Smith (2016), una aplicación monolítica típica, consiste de al menos tres capas:

- Presentación: son componentes que gestionan solicitudes HTTP e implementan una interfaz de usuario sofisticada.
- Lógica de negocio: componentes que constituyen el núcleo de la aplicación e implementan la lógica de negocio[12].
- Acceso a datos: componentes que interactúan con la infraestructura, como bases de datos o intermediarios de mensajes.

1.3.2 Ventajas

La arquitectura monolítica, ofrece las siguientes ventajas:

- Simplicidad: es simple diseñar, desarrollar y mantener en comparación con arquitecturas distribuidas. Todo el código y la lógica del negocio están en un único lugar, lo que facilita la comprensión y la depuración del sistema.
- Facilidad de desarrollo: resulta adecuada para equipos pequeños o proyectos con recursos limitados. No requiere la coordinación compleja entre diferentes equipos, lo que simplifica el proceso de desarrollo.
- Despliegue sencillo: el despliegue resulta directo, ya que todo el código se despliega como una sola unidad.
- Menor latencia: el hecho de que todas las funciones y procesos se ejecuten en una misma aplicación, contribuye a disminuir el tiempo de latencia[13].
- Facilidad para la depuración y pruebas: al estar todo el código en un solo lugar, la depuración y las pruebas resultan ser más sencillas y directas.
- Menor costo: inicialmente, el desarrollo y la implementación de una aplicación monolítica implica un costo inferior al de construir una arquitectura distribuida, dado que demanda infraestructura reducida y menor complejidad en el diseño.

1.3.3 Desventajas

Con el transcurso del tiempo y a medida que el sistema crece en tamaño y complejidad, la arquitectura monolítica presenta las siguientes desventajas.

[12] Lógica de negocio: conjunto de reglas, procesos y procedimientos que determinan cómo una aplicación procesa y transforma la información para cumplir con los objetivos de la organización

[13] Latencia: es una medida de los retardos de un sistema. Ej. la latencia de las redes, es el tiempo que tardan los datos en recorrer de un punto a otro de la red.

- Dificultad para escalar: solo se puede escalar de forma vertical, lo que requiere aumentar los recursos del servidor donde se encuentra instalada la aplicación.
- Fuerte acoplamiento: los componentes de una aplicación monolítica suelen estar estrechamente acoplados, lo que significa que un cambio en una funcionalidad o parte de la aplicación puede afectar a otros componentes, provocando fallas, haciéndolo difícil de mantener y actualizar.
- Dificultad para adoptar nuevas tecnologías**:** cuando una aplicación está desarrollada íntegramente con una sola tecnología, resulta complejo incorporar nuevas tecnologías o lenguajes de programación, ya que cualquier cambio significativo suele requerir la reescritura total de la aplicación.
- Complejidad creciente: a medida que la aplicación aumenta en tamaño y funcionalidad, la arquitectura monolítica se vuelve progresivamente más difícil de comprender y mantener. Esto complica la refactorización del código y genera un mayor grado de acoplamiento entre sus componentes.
- Dificultad para la colaboración: en equipos grandes, se hace difícil que varios desarrolladores trabajen en una aplicación monolítica al mismo tiempo.
- Mayor riesgo de fallos: debido al fuerte acoplamiento y a la falta de aislamiento entre los componentes, una falla en una parte de la aplicación puede llevar a la caída de toda la aplicación.

1.4 Arquitectura de microservicios

El término microservicios, surgido entre los años 2011 y 2014, se refiere a un enfoque de diseño de aplicaciones de software basado en servicios pequeños, autónomos y desplegables de forma independiente. Cada microservicio se centra en una funcionalidad específica, lo que permite desarrollar, mantener y escalar cada componente de manera separada. Esta arquitectura facilita la adaptación a cambios, mejora la resiliencia del sistema y permite combinar tecnologías diferentes según las necesidades de cada servicio.

1.4.1 Microservicios

Lewis & Fowler (2014), introdujeron el concepto a través de la publicación del artículo *Microservices*. Allí lo definieron como un estilo arquitectónico para el desarrollo de aplicaciones, basado en un conjunto de pequeños servicios que se ejecutan en procesos propios y se comunican mediante mecanismos ligeros, generalmente a través de APIs HTTP. Cada uno de estos módulos representa una unidad funcional concreta e independiente, que en conjunto conforman la funcionalidad completa de una aplicación.

Este enfoque permite desacoplar la lógica de negocio en elementos autónomos que interactúan entre sí. Dichos elementos suelen estar implementados en distintos

lenguajes de programación y utilizar tecnologías de almacenamiento diversas, tanto bases de datos relacionales como no relacionales. Debido a su bajo nivel de acoplamiento y carácter distribuido, resultan fácilmente escalables y desplegables. Con estas características, el paradigma ha ganado gran popularidad en la comunidad de desarrollo de *software*.

De acuerdo con Daya et al. (2015), este modelo arquitectónico permite que grandes y complejas aplicaciones se construyan a partir de servicios implementables de manera independiente y con un acoplamiento débil. Cada componente se centra en realizar una sola tarea, representando una capacidad empresarial específica.

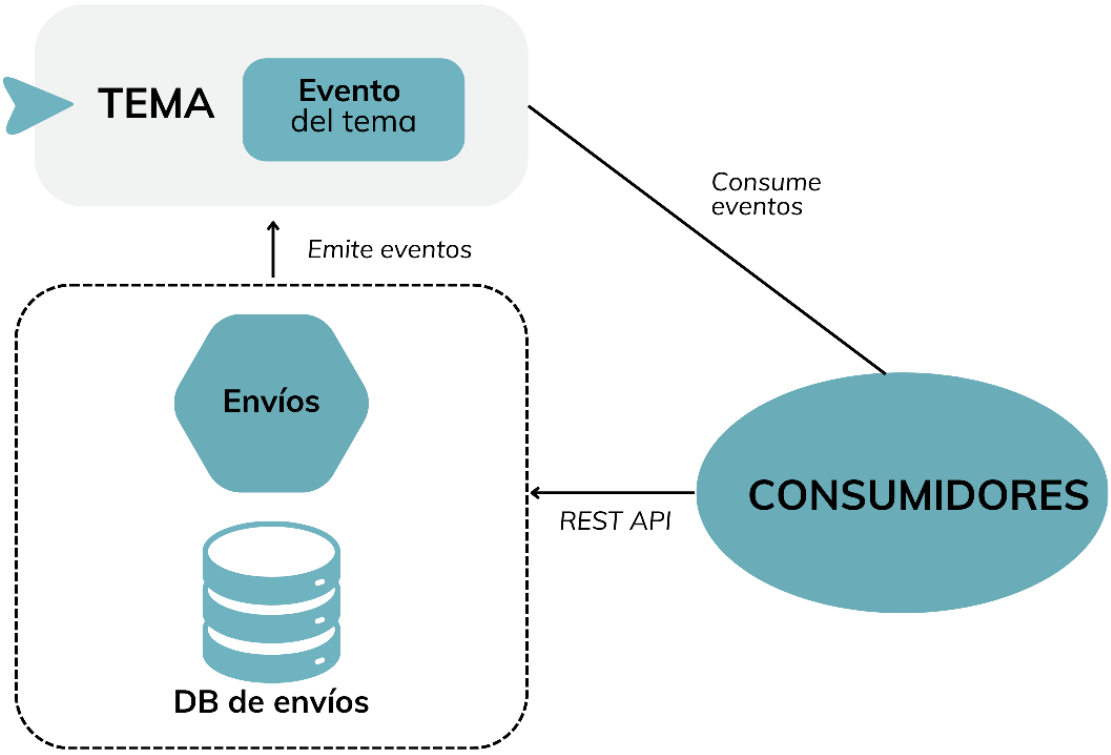
La comunicación entre ellos se realiza mediante interfaces de programación de aplicaciones (API), dentro de un contexto acotado, sin necesidad de conocer la implementación interna de los demás. Bonér (2016), describe esta arquitectura como un sistema conformado por servicios pequeños y aislados, propietarios de sus propios datos, independientes, escalables y resistentes a fallos. Estos se integran para constituir un sistema cohesivo y flexible, superando en adaptabilidad a los sistemas empresariales tradicionales.

Por su parte, Newman (2021), explica que los servicios se modelan en torno a dominios de negocio, encapsulan funcionalidad y la exponen a otros a través de la red. De esta manera, un sistema complejo puede construirse a partir de componentes básicos: por ejemplo, un servicio puede gestionar inventario, otro las órdenes de compra y otro los envíos, conformando juntos un sistema de comercio electrónico completo.

Asimismo, esta arquitectura orientada a servicios enfatiza la definición de límites claros y la capacidad de despliegue independiente. Cada componente se concibe como una “caja negra” que alberga la funcionalidad empresarial en puntos finales de red, como una cola o una API REST (Ver Figura 3). Los consumidores acceden a través de esas interfaces sin conocer los detalles internos de implementación, los cuales se mantienen ocultos. Esta ocultación permite modificar libremente la lógica interna siempre que las interfaces externas se mantengan compatibles con versiones anteriores. De este modo, los cambios dentro de los límites de un servicio no deberían impactar en los consumidores ascendentes, lo que habilita la liberación independiente de funcionalidades. Contar con límites estables y bien definidos genera sistemas con acoplamiento flexible y mayor cohesión, favoreciendo tanto la evolución tecnológica como la capacidad de adaptación a nuevas necesidades.

Figura 3

Microservicio exponiendo su funcionalidad a través de una API REST y un tópico



Nota. Adaptada de Newman (2019). *Building microservices: Designing fine-grained systems*. O'Reilly Media

1.4.2 Características de la arquitectura de microservicios

- Componentización: los sistemas están divididos en pequeños servicios independientes que pueden ser desarrollados, desplegados y escalados de forma independiente. Cada servicio es una unidad autónoma que se enfoca en una tarea específica del negocio.
- Organización alrededor del negocio: los servicios están diseñados para reflejar la estructura organizativa o funcional del negocio. Cada servicio se enfoca en una función o proceso de negocio claramente definido. Surgen los equipos multifuncionales.
- Descentralización de datos y gestión: cada servicio es responsable de su propia base de datos y gestiona sus propios datos. Esto promueve la independencia y la escalabilidad de los servicios, pero también introduce desafíos en términos de consistencia de datos y coordinación entre servicios.
- Comunicación a través de APIs: los servicios se comunican entre sí a través de interfaces bien definidas, generalmente a través de APIs (Interfaces de Programación

de Aplicaciones). Esto permite que los servicios se comuniquen de manera efectiva y promueve la flexibilidad y la interoperabilidad.

- Escalabilidad y elasticidad: los servicios pueden ser escalados de forma independiente según sea necesario para manejar diferentes cargas de trabajo. Esto permite una gestión más eficiente de los recursos y una mejor adaptación a las fluctuaciones en la demanda.
- Despliegue automatizado: se hace uso extensivo de prácticas de despliegue automatizado, como la integración continua y la entrega continua (CI/CD), para permitir la implementación rápida y confiable de nuevos servicios y actualizaciones.
- Resiliencia y tolerancia a fallos: los sistemas de microservicios están diseñados para ser resistentes a fallos. Cada servicio es independiente y aislado, lo que limita el impacto de los fallos en otras partes del sistema y permite una recuperación más rápida.
- Orientación a la evolución: los sistemas de microservicios están diseñados para ser adaptables y evolucionar con el tiempo. Los servicios pueden ser actualizados, reemplazados o reorganizados según sea necesario para satisfacer las cambiantes necesidades del negocio.

1.4.3 Ventajas

- Despliegue independiente: los microservicios permiten a los equipos de desarrollo desplegar componentes de manera independiente unos de otros. Esto significa que se pueden lanzar nuevas versiones de servicios sin afectar a otros componentes del sistema. Facilita la implementación continua y la entrega rápida de nuevas funcionalidades, ya que no es necesario esperar a que todo el sistema sea desarrollado y probado en su totalidad.
- Escalabilidad: los microservicios permiten escalar partes específicas de una aplicación de manera independiente, lo que facilita la gestión de la carga y el rendimiento. Esto significa que es posible asignar más recursos a los servicios que requieren mayor capacidad, mientras que otros servicios permanecen con recursos mínimos si no experimentan una alta demanda.
- Adaptación a nuevas tecnologías: dado que cada microservicio es una entidad independiente, los equipos seleccionan las tecnologías más adecuadas para cada uno, lo que permite la experimentación y la adopción de nuevas herramientas sin afectar al sistema en su conjunto. Esto también facilita la actualización de tecnologías obsoletas o la incorporación de soluciones más modernas en partes específicas del sistema.

- Facilidad en el desarrollo y mantenimiento: al dividir una aplicación en componentes más pequeños y manejables, los equipos pueden trabajar de manera más eficiente y centrarse en áreas específicas de la aplicación. Esto simplifica las pruebas, ya que los microservicios pueden ser probados de forma aislada, y el mantenimiento se vuelve más manejable al tener componentes más pequeños y específicos.
- Resiliencia y tolerancia a fallos: al diseñar cada servicio como una entidad independiente, los fallos en un componente no afectan necesariamente a otros dentro del sistema. Esto mejora la resiliencia general, ya que los errores se aíslan y se gestionan de manera más efectiva.
- Facilidad en la gestión del equipo: los microservicios son desarrollados y mantenidos por equipos pequeños y especializados, lo que facilita la gestión y la comunicación dentro de esos grupos de trabajo. Además, esto favorece una mayor agilidad en el desarrollo, ya que los equipos trabajan de manera independiente y rápida en sus respectivos servicios.
- Mayor adaptabilidad al cambio: la arquitectura de microservicios permite a una organización adaptarse más fácilmente a los cambios en los requisitos del negocio, ya que las actualizaciones y las nuevas características se implementan de manera más rápida y con menor riesgo. Esto proporciona una ventaja competitiva al permitir que las organizaciones respondan con agilidad a las demandas del mercado y a las necesidades de los usuarios.

1.4.4 Desventajas

- Complejidad en la gestión de la red: al dividir una aplicación en múltiples servicios independientes, se introduce una mayor complejidad en la red de comunicación entre estos servicios. Esto genera una sobrecarga de la red y plantea desafíos adicionales en la gestión y el monitoreo de la comunicación entre los servicios.
- Desafíos en la consistencia de datos: la descentralización de datos introduce dificultades en términos de consistencia y sincronización entre servicios. Mantener la coherencia de la información a través de múltiples servicios resulta complejo y exige estrategias específicas, como la implementación de patrones de consistencia eventual.
- Dificultades en la transacción distribuida: las transacciones que involucran múltiples servicios distribuidos son más difíciles de gestionar que las transacciones monolíticas. Coordinar transacciones distribuidas y garantizar la consistencia de datos en este contexto resulta complejo y demanda estrategias avanzadas, como el uso de patrones de compensación.

- Mayor complejidad en el desarrollo y la prueba: la naturaleza distribuida de los microservicios introduce una mayor complejidad en el desarrollo y la validación de los sistemas. Coordinar cambios en múltiples servicios y garantizar la compatibilidad entre ellos requiere una planificación cuidadosa y un enfoque sólido en la gestión de dependencias.
- Mayor sobrecarga operativa: la gestión de múltiples servicios distribuidos introduce una mayor sobrecarga operativa en términos de implementación, monitoreo, escalado y resolución de problemas. Esto requiere inversiones adicionales en herramientas y procesos de operaciones.
- Desafíos en la seguridad: la seguridad en entornos de microservicios resulta más compleja que en sistemas monolíticos, especialmente en lo referente a la gestión de identidad y acceso, así como en la protección contra amenazas como la inyección de código malicioso y la escalada de privilegios.
- Mayor costo de desarrollo inicial: la adopción de una arquitectura de microservicios demanda una inversión significativa en términos de desarrollo inicial, infraestructura y habilidades técnicas. Este aspecto representa un desafío para las organizaciones con recursos limitados o que inician desde cero.

En síntesis, el conocimiento de los sistemas *legacy*, los ERP y las arquitecturas monolíticas y de microservicios proporciona el marco teórico necesario para abordar el proceso de transformación tecnológica. A partir de esta base, el siguiente capítulo se enfoca en las herramientas y tecnologías que hacen posible llevar a la práctica dicha transición, analizando *stacks* tecnológicos, contenedores, orquestación y computación en la nube.

Capítulo II – Tecnologías y patrones de diseño

Este capítulo desarrolla los conceptos fundamentales que constituyen la base para comprender y sustentar el proceso de transición desde una arquitectura monolítica hacia una arquitectura orientada a microservicios. En primer lugar, se realiza una introducción a los *stacks* tecnológicos, concebidos como el conjunto de herramientas, lenguajes y marcos de trabajo que permiten construir, desplegar y mantener aplicaciones modernas de manera eficiente.

Posteriormente, se aborda el rol de los contenedores como mecanismo esencial para garantizar la portabilidad y el aislamiento de los servicios, junto con la orquestación de contenedores, indispensable para gestionar la escalabilidad, disponibilidad y resiliencia en entornos distribuidos. En esta misma línea, se introducen los principios de integración y entrega continua, los cuales constituyen prácticas fundamentales para lograr ciclos de desarrollo más ágiles, confiables y automatizados.

Asimismo, se examinan los conceptos clave de la computación en la nube, considerando sus modelos de servicio e implementación como pilares que posibilitan la elasticidad, la reducción de costos y la optimización de los recursos tecnológicos. Finalmente, se describen patrones de diseño que resultan relevantes en el contexto de sistemas distribuidos, incluyendo aquellos que se aplican de forma específica a arquitecturas basadas en microservicios.

2.1 Stack tecnológico

Invgate (2025)[14] define un *stack* tecnológico como el conjunto de tecnologías, entre las que se incluyen lenguajes de programación, *frameworks*[15], bases de datos, herramientas y servicios, que se integran para diseñar, construir y operar aplicaciones o sistemas. Estos elementos trabajan de manera conjunta para proporcionar una solución funcional y garantizar una experiencia de usuario consistente y uniforme en todas las funcionalidades de la aplicación. A continuación, se presenta un modelo de *stack* tecnológico:

- Lenguajes de Programación: utilizados para implementar la lógica del negocio y las funcionalidades de la aplicación. Ejemplos comunes son *Java*, *Python*, *JavaScrip*t (*Node.js*), *Go*, entre otros.
- Frameworks y Librerías: conjuntos de herramientas y librerías que facilitan el desarrollo de software dentro del lenguaje elegido. Por ejemplo, *Spring Boot* para *Java*, *Flask* o *Django* para *Python*, *Express.js* para *Node.js*, *Vuetify* para *Vue.js.*

[14] Invgate es una empresa especializada en soluciones de *software* para la gestión de servicios de tecnología de la información (ITSM) y gestión de activos de TI (ITAM).

[15] *Framework*: es un conjunto estructurado de herramientas, bibliotecas y componentes de *software* que proporciona una base predefinida para desarrollar aplicaciones de manera más eficiente y consistente.

- Base de Datos: los sistemas de gestión de bases de datos (DBMS) se utilizan para almacenar y manipular datos. Estos incluyen bases de datos SQL (relacionales) como *MySQL*, *PostgreSQL* o *SQL Server*, así como bases de datos *NoSQL* (no relacionales) como *MongoDB* y *Redis*.
- Infraestructura y Entorno de Ejecución: incluye el sistema operativo, el servidor *web*, el servidor de aplicaciones, los contenedores y la orquestación de contenedores.
- Herramientas de Desarrollo y Gestión: IDEs (*Integrated Development Environment*), sistemas de control de versiones (Git), herramientas de integración continua, despliegue continuo (CI/CD), herramientas de monitoreo y *logging*, entre otras.
- Servicios en la Nube: en algunos casos, el stack tecnológico puede incluir servicios en la nube como AWS (*Amazon Web Services*), *Microsoft Azure*, *Google Cloud Platform*, *Oracle Cloud Infrastructure*, que proporcionan infraestructura escalable y servicios gestionados.
- Servicios de mensajerías
    - Message Queues (colas de mensajes): *RabbitMQ*, *Apache Kafka* y *Amazon SQS*. Estas plataformas permiten que los microservicios publiquen mensajes en colas y los consuman según sea necesario, facilitando la comunicación asíncrona y la gestión de carga.
    - Message Brokers: como *Redis*, que se utiliza para el intercambio de mensajes en tiempo real o para la implementación de sistemas de notificación.

2.2 Contenedores

Según Spartan Cybersecurity (2025),[16] son descritos como unidades ligeras y portátiles que encapsulan aplicaciones junto con todas sus dependencias necesarias para ejecutarse de forma consistente en diferentes entornos. Un contenedor, a diferencia de una máquina virtual, no requiere un sistema operativo completo, lo que lo vuelve más eficiente. A continuación se mencionan las características de los contenedores:

- Separación de responsabilidades: los desarrolladores se centran en escribir el código y definir las dependencias de la aplicación dentro del contenedor. Esto incluye bibliotecas, *frameworks* y configuraciones específicas necesarias para que la aplicación funcione correctamente.
- Facilita el despliegue: los equipos de operaciones (*DevOps*) toman esos contenedores y los despliegan fácilmente en diferentes entornos, como desarrollo, pruebas y producción. Los contenedores aseguran que la aplicación funcionará de la misma manera en cualquier entorno donde se despliegue, eliminando problemas de "funciona en mi máquina" o incompatibilidades.

[16] Spartan Cybersecurity es una academia de educación en línea especializada en ciberseguridad y hacking ético.

- Portabilidad: los contenedores encapsulan todo lo que una aplicación necesita para ejecutarse, incluidas las bibliotecas y configuraciones, generando aplicaciones altamente portátiles, lo que permite ejecutarlas en diferentes plataformas (*Windows*, *Linux*) y en la nube sin cambios significativos.
- Versionado y control de versiones: los contenedores permiten versionar aplicaciones y sus dependencias de una manera controlada y reproducible. Esto facilita la gestión de versiones de software y asegura que las aplicaciones se desplieguen con las versiones correctas de las bibliotecas y configuraciones necesarias.
- Consistencia: una vez empaquetada[17] en una imagen de contenedor, una aplicación siempre se ejecutará de la misma manera. Esta consistencia simplifica las pruebas, el despliegue y el escalado de aplicaciones.
- Eficiencia: los contenedores se consideran eficientes debido a que comparten el *kernel* del sistema operativo del host, evitando la necesidad de incluir un sistema operativo completo en cada instancia. Esta característica permite un uso más eficiente de los recursos computacionales, optimizando la memoria, el almacenamiento y el tiempo de despliegue en comparación con las máquinas virtuales tradicionales.

En resumen, los contenedores ofrecen una solución eficiente para el desarrollo y despliegue de aplicaciones al separar claramente las actividades entre los equipos de desarrollo y operaciones, facilitando la colaboración y la consistencia en todo el ciclo de vida de la aplicación.

2.3 Orquestación de contenedores

Según *Amazon Web Services* (2025)[18], un orquestador[19] permite automatizar tareas como la gestión de redes y recursos necesarios para ejecutar aplicaciones a gran escala. Mediante la contenerización[20], se agrupan aplicaciones junto con sus dependencias para garantizar su portabilidad. A medida que las arquitecturas basadas en microservicios crecen, la cantidad de contenedores se multiplican significativamente. Por ello, las herramientas de orquestación ayudan a simplificar el manejo de esta

[17] El término **"**empaquetada**"** se refiere a una aplicación que se entrega junto con todas sus dependencias necesarias (bibliotecas, configuraciones, archivos de soporte, etc.) en un único paquete que puede ejecutarse de manera consistente en cualquier entorno compatible.

[18] Amazon Web Services (AWS**)** es una plataforma integral de servicios en la nube proporcionada por Amazon, que ofrece más de 200 servicios globales bajo demanda, incluyendo computación, almacenamiento, bases de datos, redes, inteligencia artificial, análisis de datos, Internet de las Cosas (IoT), entre otros.

[19] Kubernetes es uno de los orquestadores de contenedores más populares y ampliamente utilizados en la actualidad. Fue desarrollado originalmente por *Google* y ahora es mantenido por la *Cloud Native Computing Foundation (CNCF)*

[20] Contenerización: es el proceso de empaquetar una aplicación junto con todas sus dependencias, configuraciones y librerías necesarias dentro de un contenedor, de manera que pueda ejecutarse de forma consistente en cualquier entorno que soporte contenedores.

infraestructura, automatizando procesos durante todo su ciclo de vida, lo que permite escalar sin aumentar la complejidad operativa.

2.4 Integración continua (CI) y Distribución continua (CD)

Integración continua (CI): Según Red Hat. (s.f.)[21], es un proceso automatizado que facilita a los desarrolladores la fusión de cambios de código en una rama principal o secundaria de manera sencilla y frecuente. Cada vez que se aplican actualizaciones, se ejecutan pasos de prueba automatizados con el fin de asegurar la confiabilidad de los cambios integrados.

El desarrollo de aplicaciones modernas busca que varios desarrolladores trabajen simultáneamente en diferentes funcionalidades de una misma aplicación. No obstante, si una organización decide unificar todo el código diverso en un solo día —práctica conocida como "día de la fusión"—, las tareas pueden volverse tediosas, manuales y extremadamente lentas. Esto se debe a que las modificaciones realizadas en paralelo pero de forma aislada pueden generar conflictos entre sí.

La integración continua se plantea como una solución a la proliferación de múltiples ramificaciones en una aplicación, las cuales pueden entrar en conflicto. Su implementación es efectiva cuando las modificaciones se integran, se validan mediante la compilación automática de la aplicación y se someten a diversas pruebas automatizadas, principalmente de unidad e integración, para verificar que no se hayan introducido errores. Así, se evalúan aspectos que van desde las clases y su funcionamiento hasta los distintos módulos de la aplicación. Una de las principales ventajas de la integración continua es que, si durante las pruebas automatizadas se detecta un conflicto entre el nuevo código y el existente, este puede ser corregido de manera rápida y sencilla.

Distribución continua (CD): Según Red Hat. (s.f.), consiste en automatizar el proceso mediante el cual el código validado queda listo para ser desplegado en producción.

El objetivo central es garantizar que la base de código se mantenga en todo momento en condiciones de ser liberada. De este modo, cualquier cambio aprobado puede trasladarse rápidamente a un entorno productivo sin necesidad de procesos manuales extensos. Esta práctica permite a los equipos de operaciones implementar nuevas versiones de manera ágil, reduciendo tiempos de entrega y minimizando riesgos asociados a errores humanos.

En la práctica, implica que las modificaciones desarrolladas se someten a pruebas automatizadas, se almacenan en repositorios y quedan preparadas para su despliegue inmediato. De esta manera, se facilita la incorporación continua de mejoras y nuevas

[21] Red Hat es una empresa multinacional estadounidense especializada en soluciones de software de código abierto para entornos empresariales. Fundada en 1993 por Marc Ewing y Bob Young.

funcionalidades, asegurando que la entrega de software sea eficiente, confiable y alineada con las necesidades del negocio.

2.5 Computación en la nube

Según Santana (2023), representa una evolución fundamental en el desarrollo de soluciones tecnológicas modernas. Inspirada en los lineamientos del Instituto Nacional de Estándares y Tecnología (NIST)[22], la nube permite el acceso bajo demanda a recursos informáticos configurables, como redes, servidores, almacenamiento y aplicaciones, que pueden ser gestionados con mínima intervención del proveedor.

En este contexto, la computación en la nube se ha consolidado como un paradigma que transforma la manera en que las organizaciones diseñan, despliegan y consumen servicios tecnológicos. Su flexibilidad y escalabilidad permiten a las empresas responder con rapidez a los cambios en el mercado y optimizar sus costos operativos mediante modelos de pago por uso. Asimismo, posibilita la innovación constante, al brindar acceso a tecnologías avanzadas sin necesidad de grandes inversiones iniciales en infraestructura.

2.5.1 Tipos de nube

Existen tres tipos de nube: pública, privada e híbrida. Cada una de ellas demanda un grado particular de gestión por parte del cliente y proporciona niveles diferenciados de seguridad, de acuerdo con sus características y ámbito de aplicación.

Nube pública: toda la infraestructura informática está ubicada en las instalaciones del proveedor de servicios, quién presta servicios al cliente a través de Internet. Los clientes no tienen que preocuparse por mantener sus recursos informáticos y pueden agregar rápidamente más usuarios o potencia informática, según sea necesario. En este modelo, diversos clientes comparten la infraestructura informática del proveedor de la nube.

Nube privada: es de uso exclusivo de una organización. Se puede alojar en la ubicación de la organización o en el centro de datos del proveedor de la nube. Proporciona mayor seguridad y control.

Nube híbrida: es una combinación de nubes públicas y privadas. En general, los clientes alojan sus aplicaciones críticas en sus propios servidores para mayor seguridad y control, y las aplicaciones secundarias se almacenan en un entorno proporcionado por el proveedor de la nube.

2.5.2 Servicios de la nube

Existen tres modelos de servicios en la nube: software como servicio (*software as a service,* SaaS), plataforma como servicio (*plataform as a service,* PaaS) e infraestructura como servicio (*infrastructure as a service,* IaaS).

[22] NIST: (por sus siglas en inglés, *National Institute of Standards and Technology*) es el Instituto Nacional de Estándares y Tecnología de los Estados Unidos.

SaaS: el software como servicio (SaaS), es un modelo de implementación de software en el que el proveedor de la nube aloja las aplicaciones del cliente en su entorno de nube. El cliente accede a sus aplicaciones a través de Internet. En lugar de pagar y mantener su propia infraestructura de computación, los clientes de SaaS aprovechan la suscripción al servicio de pago por consumo.

Este modelo permite incorporar rápidamente tecnología innovadora y mantener las aplicaciones actualizadas mediante procesos automáticos, reduciendo así la carga sobre los recursos internos. Además, ofrece la posibilidad de ampliar o reducir los servicios según las fluctuaciones de las cargas de trabajo, incorporando nuevas funciones o capacidades conforme sea necesario.

PaaS: la plataforma como servicio (PaaS), ofrece a los clientes la ventaja de acceder a las herramientas para desarrolladores que necesitan a fin de crear y gestionar aplicaciones móviles y web sin tener que invertir en la infraestructura subyacente ni mantenerla. El proveedor aloja los componentes de infraestructura y *middleware*, y el cliente accede a esos servicios a través de un navegador *web*.

Para mejorar la productividad, las soluciones PaaS cuentan con componentes de programación listos para usar, que permiten a los desarrolladores integrar nuevas características en sus aplicaciones, incluidas tecnologías innovadoras como la inteligencia artificial (IA), *chatbots*, cadena de bloques e Internet de las cosas.

IaaS: la infraestructura como servicio (IaaS), permite a los clientes acceder a servicios de infraestructura a través de Internet, según la demanda. La principal ventaja es que el proveedor de servicios en la nube aloja los componentes de infraestructura que proporcionan capacidad de red, almacenamiento y computación para que sus suscriptores puedan ejecutar sus cargas de trabajo en la nube. El suscriptor de la nube generalmente es responsable de instalar, configurar, proteger y mantener todo el *software* de la solución nativa de nube, como la base de datos, el *middleware* y el *software* de aplicaciones.

2.5.3 Beneficios de la computación en la nube para las arquitecturas de microservicios

Según Santana (2023), la computación en la nube representa un entorno altamente favorable para el desarrollo, despliegue y mantenimiento de arquitecturas de microservicios. Uno de los principales beneficios es la provisión bajo demanda de recursos de cómputo, almacenamiento y red, que permite escalar de forma independiente cada microservicio según las necesidades de carga. Esto se traduce en un uso eficiente de los recursos y en una elasticidad que es difícil de lograr en entornos tradicionales *on-premises.*

Además, el enfoque *cloud* facilita la adopción de contenedores y orquestadores (capítulo 2.2 y capítulo 2.3), que permiten encapsular los microservicios junto con sus dependencias, facilitando la portabilidad, la replicación y el aislamiento de fallos. Estas características son esenciales para una arquitectura distribuida y desacoplada, donde cada componente puede evolucionar, escalar o reiniciarse sin afectar al resto del sistema.

La computación en la nube posibilita la implementación de prácticas de integración continua y despliegue continuo de manera eficiente. Los entornos reproducibles y automatizados proporcionados por los principales proveedores *cloud,* permiten a los equipos de desarrollo automatizar pruebas, compilaciones y despliegues mediante *pipelines* que aseguran una entrega constante de valor. En una arquitectura de microservicios, caracterizada por cambios frecuentes y localizados, esta capacidad resulta esencial para mantener la agilidad sin comprometer la estabilidad del sistema.

Un beneficio de relevancia es la tolerancia a fallos. Los servicios en la nube están diseñados para operar en entornos distribuidos y redundantes, donde los fallos de *hardware*, red o *software* son esperados y manejados de forma automatizada. Los microservicios pueden ser desplegados en múltiples zonas de disponibilidad, y los balanceadores de carga redirigen el tráfico automáticamente en caso de que un nodo falle. Este enfoque reduce significativamente el tiempo de inactividad y mejora la disponibilidad general del sistema.

2.6 Patrones de diseño

Los patrones de diseño, definidos como soluciones reutilizables y probadas a problemas recurrentes en el desarrollo de software, Gamma et al (1994), adquieren un papel fundamental en procesos de modernización tecnológica. En particular, durante la migración de sistemas monolíticos hacia arquitecturas basadas en microservicios (capítulo 1.3 y capítulo 1.4). Estos patrones ofrecen lineamientos que permiten enfrentar de manera estructurada los desafíos asociados a la descomposición funcional, la comunicación distribuida y la gestión de la complejidad inherente a este tipo de transición.

La separación de un sistema monolítico (capítulo 1.3.1) en microservicios (capítulo 1.4.1), implica desacoplar funcionalidades previamente integradas en un único bloque, con el objetivo de obtener servicios autónomos y escalables. En este contexto, los patrones de diseño proporcionan un marco conceptual y práctico que guía las decisiones arquitectónicas y reduce los riesgos asociados al cambio. Patrones como API *Gateway* permiten centralizar la interacción entre clientes y múltiples servicios, mientras que *Circuit Breaker* incrementa la resiliencia del sistema al prevenir fallos en cascada. Asimismo, el patrón *Strangler Fig* facilita la migración progresiva de componentes, permitiendo mantener la continuidad operativa mientras nuevas funcionalidades son incorporadas bajo la lógica de microservicios.

2.6.1 Patrón Estrangulador (*Strangler Fig*)

Newman (2020),[23] describe el denominado patrón estrangulador como un enfoque conceptual orientado a implementar cambios significativos en un sistema complejo de manera incremental y controlada, con el propósito de reducir el riesgo y evitar interrupciones en el servicio. Este patrón se utiliza con frecuencia en procesos de migración de software, particularmente en la transformación de sistemas monolíticos hacia arquitecturas de microservicios o en el desacoplamiento de componentes con el fin de mejorar la escalabilidad y la flexibilidad. En este contexto, la estrategia consiste en incorporar gradualmente nuevas funcionalidades o componentes alrededor del sistema existente, lo que permite sustituir o eliminar progresivamente la infraestructura previa sin generar una disrupción considerable para los usuarios del sistema.

Este patrón se caracteriza por las siguientes ideas claves:

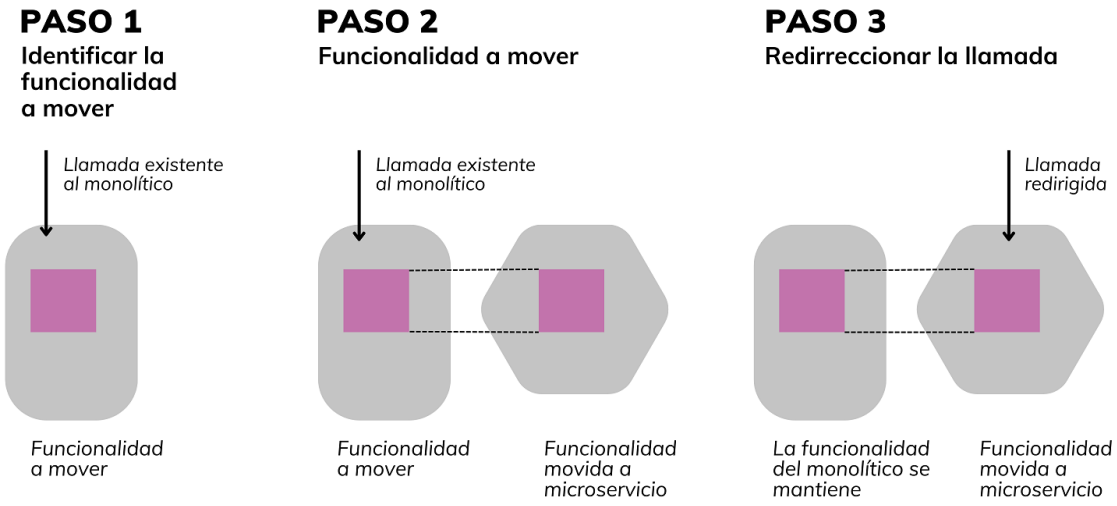
- Enfoque incremental: en lugar de rehacer todo el sistema de una vez, los cambios se introducen gradualmente. Esto reduce el riesgo de errores masivos y permite una transición más suave.
- Paralelismo y convivencia: los nuevos componentes o funcionalidades pueden coexistir con el sistema antiguo durante un tiempo, permitiendo pruebas, validaciones y ajustes sin afectar el funcionamiento general.
- Transición controlada: a medida que se introduce nueva funcionalidad, se puede redirigir el tráfico hacia estos nuevos componentes, reduciendo gradualmente la dependencia del sistema monolítico. Esto también permite que las partes antiguas del sistema sean desmanteladas en un proceso controlado.
- Flexibilidad y escalabilidad: el patrón ofrece flexibilidad para adaptarse a necesidades cambiantes y permite escalar componentes específicos sin tener que escalar todo el sistema.
- Reducción del riesgo: al ser un enfoque incremental, el patrón reduce el riesgo de fallos críticos, ya que cualquier problema suele estar contenido en la parte del sistema en transformación, sin afectar el resto del sistema.

El patrón se basa en tres pasos, identificar la funcionalidad a mover, mover la funcionalidad y por último redireccionar las llamadas. La Figura 4, muestra los tres pasos.

[23] Newman es reconocido como autor, conferencista y consultor independiente, con especial interés en temáticas vinculadas a la computación en la nube, la entrega continua y la arquitectura de microservicios. Ha participado como orador en numerosas conferencias internacionales y es autor de obras de referencia en el área, entre las que destacan *Building Microservices* y *Monolith to Microservices*, publicadas por la editorial O'Reilly.

Figura 4

Patrón estrangulador



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

2.6.2 Patrón Rama por Abstracción (*Branch By Abstraction*)

De acuerdo con Newman (2020), el patrón constituye una estrategia ampliamente utilizada en los procesos de migración de sistemas monolíticos hacia arquitecturas basadas en microservicios. Su propósito fundamental es permitir la transición gradual de componentes sin afectar la continuidad operativa del sistema. Para ello, se introduce una capa de abstracción que actúa como intermediaria entre la funcionalidad existente y la nueva implementación, garantizando que ambas puedan coexistir dentro del mismo código base mientras se completa la migración.

Este enfoque resulta particularmente útil en escenarios donde se requiere minimizar riesgos, ya que evita interrupciones en los servicios críticos y posibilita que las pruebas y validaciones se realicen de manera controlada. En este sentido, el patrón se apoya en la idea de realizar modificaciones incrementales sobre el código monolítico, de modo que la adopción de microservicios se produzca de forma progresiva y transparente para los usuarios finales.

El proceso de aplicación del patrón se desarrolla en cinco pasos:

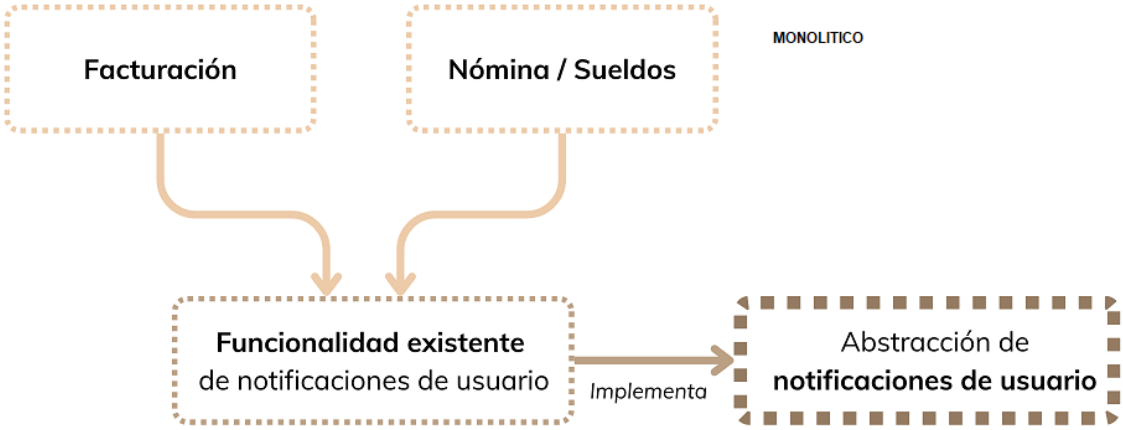
1. Creación de la abstracción: se construye una interfaz o capa intermedia que encapsula la funcionalidad que se desea migrar. Ver Figura 5.
2. Redirección de los clientes: los componentes o clientes que utilizan dicha funcionalidad son modificados para interactuar exclusivamente a través de la abstracción. Ver Figura 6.

3. Nueva implementación: se desarrolla una nueva versión de la funcionalidad, generalmente orientada a microservicios, que cumple con la misma interfaz definida en la abstracción. Ver Figura 7.
4. Sustitución progresiva: la abstracción se ajusta para dirigir las solicitudes hacia la nueva implementación, manteniendo la posibilidad de verificar su correcto funcionamiento. Ver Figura 8.
5. Depuración final: una vez validada la migración, se elimina la implementación antigua y se limpia el código, consolidando la nueva arquitectura. Ver Figura 9.

En síntesis, este patrón representa un mecanismo de transición evolutiva que facilita la modernización de sistemas *legacy*. Su valor radica en que posibilita la coexistencia controlada de versiones, reduce la complejidad del cambio y habilita a las organizaciones para avanzar hacia microservicios sin comprometer la estabilidad de la operatoria diaria. A continuación se ilustran los 5 pasos del patrón.

Figura 5

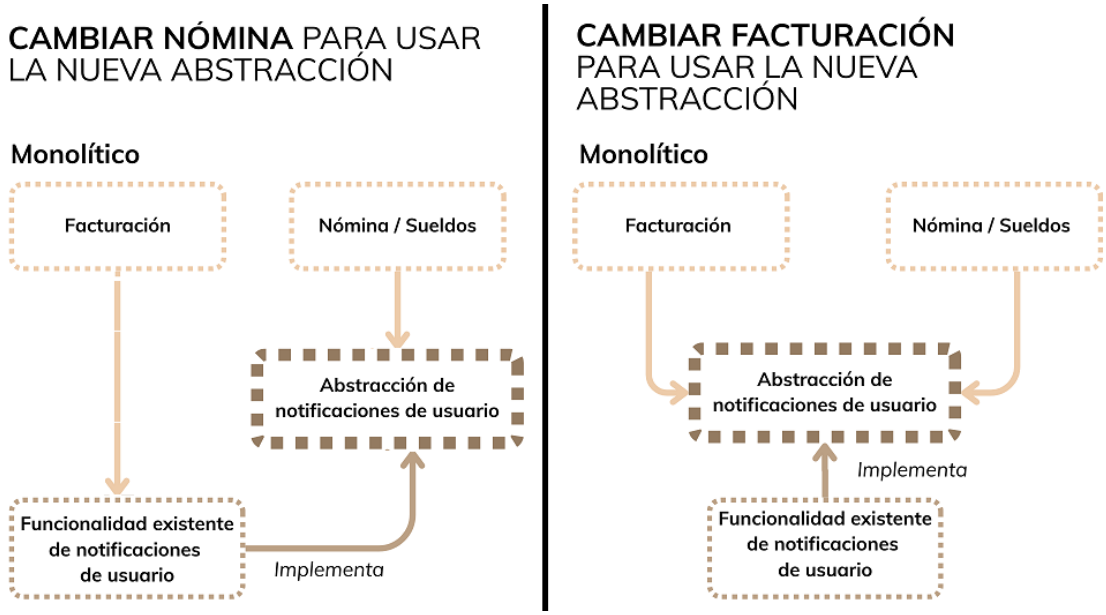
Patrón Rama por Abstracción - Paso 1



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith.* O'Reilly Media

Figura 6

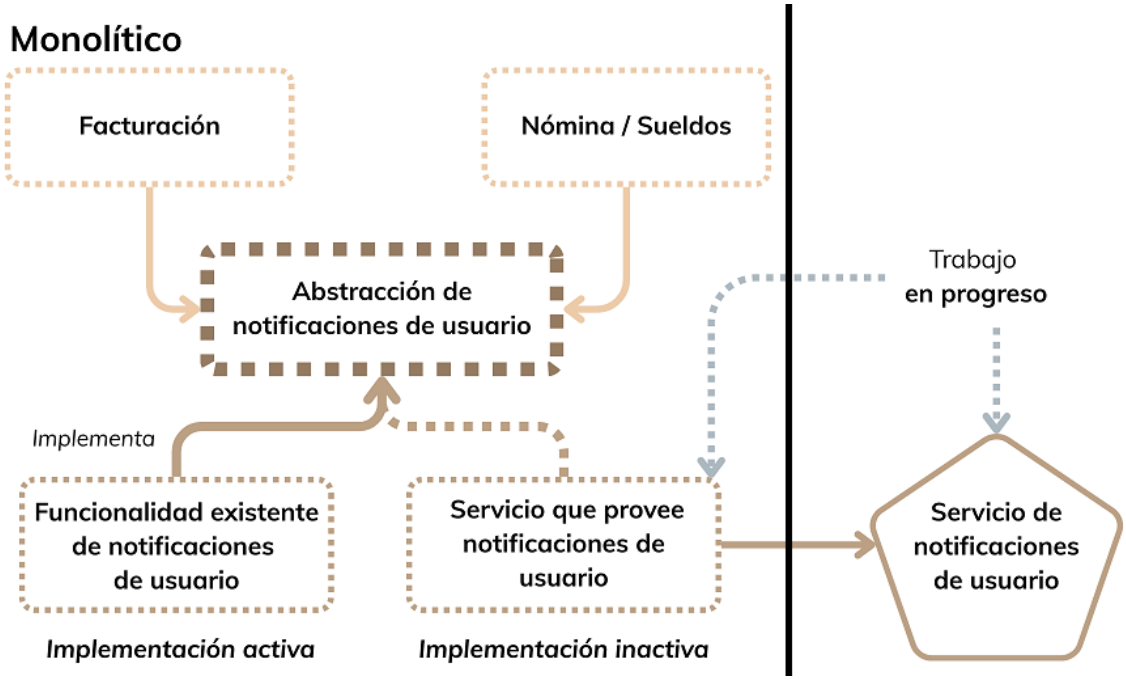
Patrón Rama por Abstracción - Paso 2



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

Figura 7

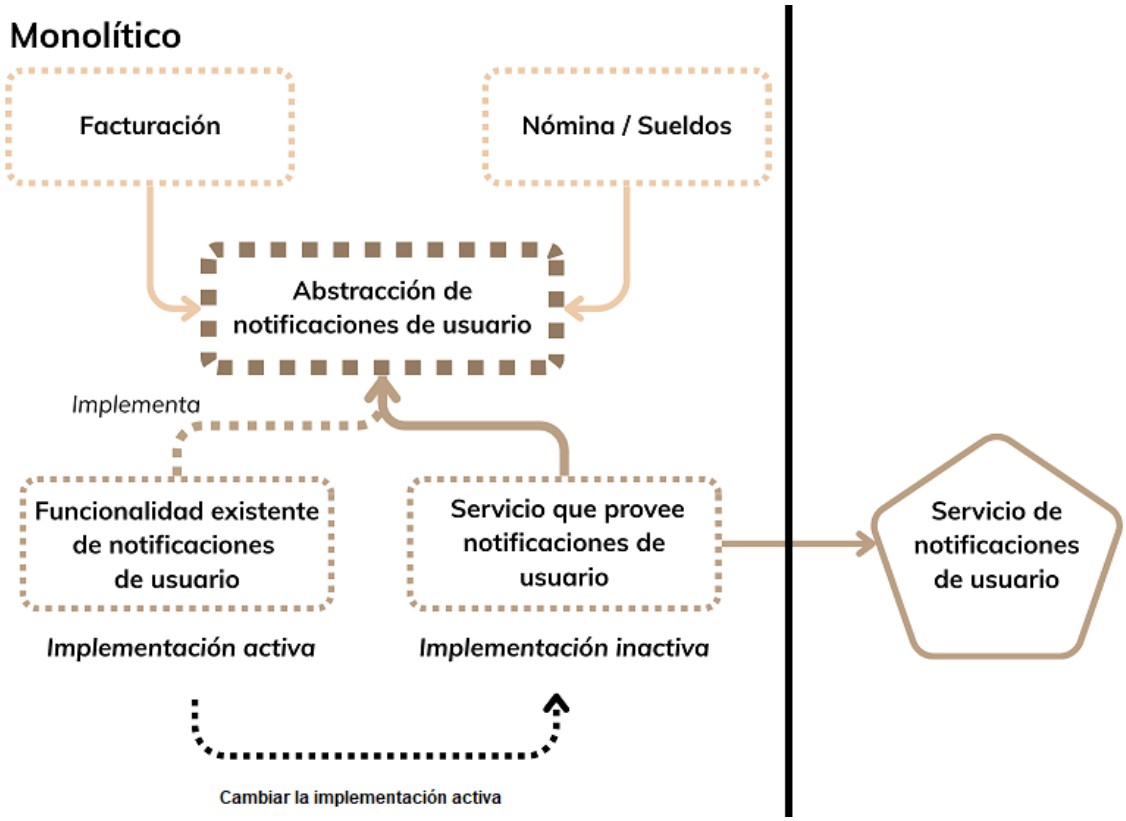
Patrón Rama por Abstracción - Paso 3



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

Figura 8

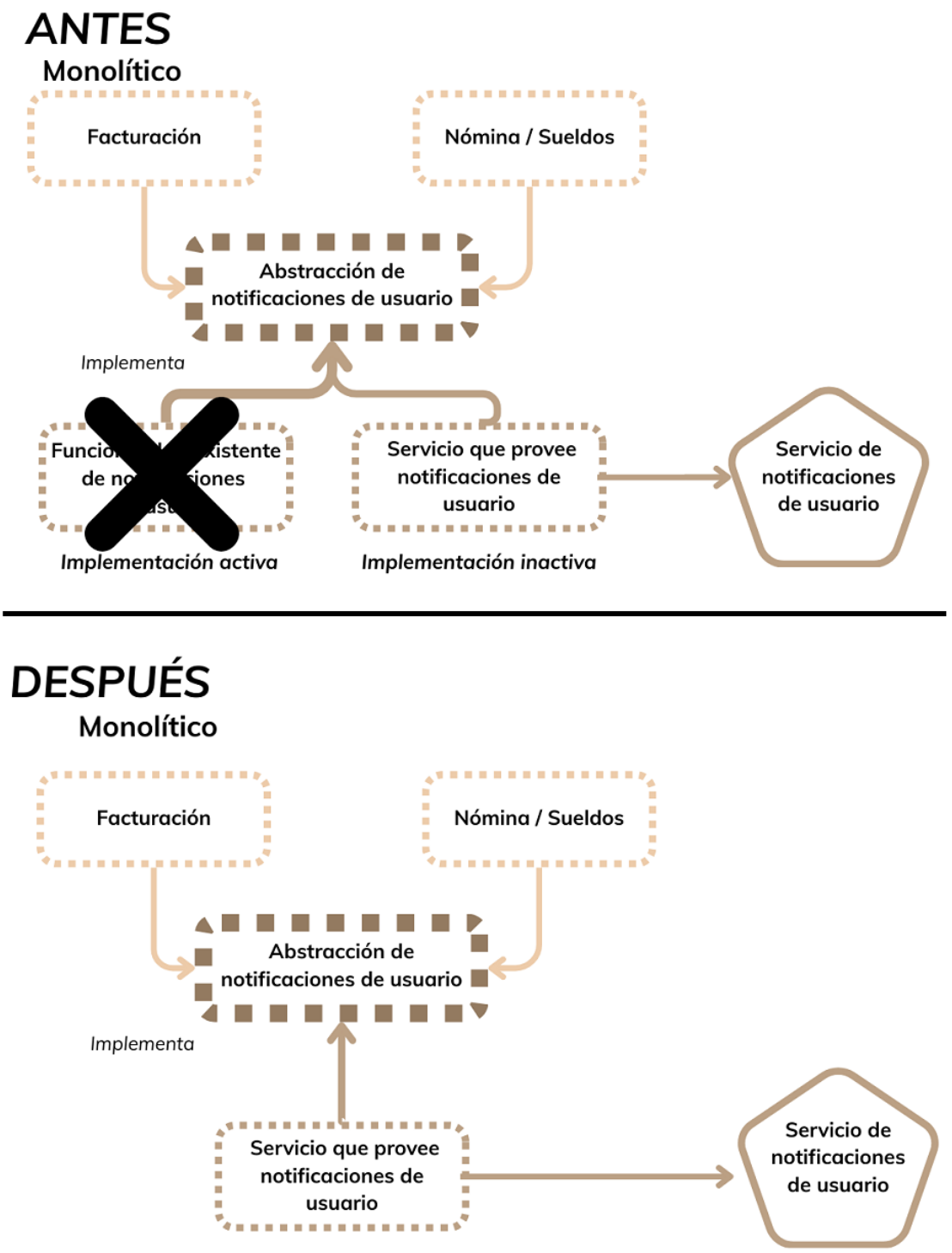
Patrón Rama por Abstracción - Paso 4



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

Figura 9

Patrón Rama por Abstracción - Paso 5



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

2.6.3 Patrón Ejecución Paralela (Paralell Run)

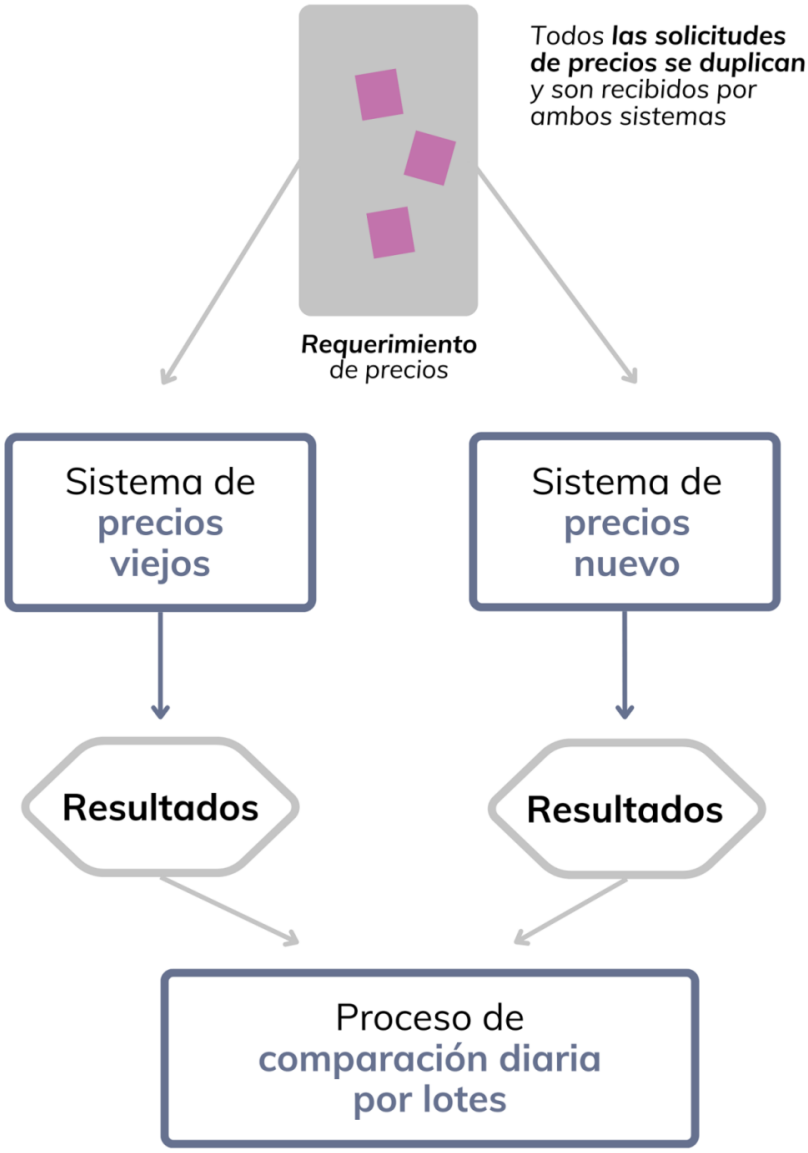
Newman (2020), este patrón es una estrategia comúnmente utilizada en el desarrollo de *software* para realizar la transición de un sistema existente a uno nuevo, de manera gradual y segura. Este enfoque implica ejecutar simultáneamente tanto el sistema existente como el nuevo durante un período de tiempo, permitiendo así probar y validar el nuevo sistema en un entorno de producción sin interrumpir el funcionamiento del sistema actual.

Tanto el patrón estrangulador como el patrón rama por abstracción, (capítulo 2.6.1 y capítulo 2.6.2), posibilitan que implementaciones antiguas y nuevas de una misma funcionalidad coexistan en un entorno de producción de manera simultánea. En estos casos, es factible disponer tanto de la versión original del sistema monolítico como de la nueva implementación basada en microservicios, aunque únicamente una de ellas se ejecuta en cada momento. Por el contrario, el patrón ejecución paralela propone la ejecución concurrente de ambas implementaciones, estableciendo una de ellas como la fuente de verdad.

Este patrón, permite comparar los resultados de ambas implementaciones para asegurar que los resultados sean equivalentes, además tiene la capacidad de determinar que la nueva implementación de la funcionalidad esté funcionando dentro de los parámetros no funcionales aceptables. En la Figura 10 se muestra un ejemplo de ejecución paralela de dos sistemas de precios y se comparan los resultados de manera offline.

Figura 10

Patrón Ejecución Paralela - Ejemplo de una ejecución paralela



Nota. Adaptada de Newman (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media

2.6.4 Patrón Puerta de Enlace (Api Gateway)

Microsoft (2022),[24] sostiene que, en arquitecturas basadas en microservicios, el patrón cumple un rol fundamental al actuar como único punto de entrada al sistema

[24] Microsoft Corporation es una empresa multinacional tecnológica con sede en Redmond, Washington, fundada en 1975. Se dedica al desarrollo y comercialización de software, servicios y hardware.

backend[25]. Este enfoque centraliza el acceso a múltiples microservicios, lo que simplifica la comunicación entre los clientes —como aplicaciones web o móviles— y los distintos servicios internos.

El beneficio principal radica en la reducción del acoplamiento entre los clientes y los microservicios, lo que permite que la estructura interna del sistema se modifique sin impactar de manera directa en los consumidores externos. Además, permite asumir responsabilidades adicionales como autenticación y autorización centralizada, registro de actividad, enrutamiento inteligente y agregación de respuestas provenientes de distintos servicios, contribuyendo así a optimizar el rendimiento y la experiencia del usuario.

Otro aspecto relevante es la capacidad de incorporar patrones de resiliencia como corta circuitos (*circuit breakers*), reintentos automáticos (*retry*) y limitación de tasas (*rate limiting*). Estas funcionalidades ofrecen un mecanismo de protección frente a sobrecargas o fallos temporales en los microservicios individuales, fortaleciendo la estabilidad global del sistema.

Asimismo, resulta especialmente valioso para la gestión de múltiples versiones de una API, la incorporación de mecanismos de cacheo[26] en respuestas recurrentes y la unificación de políticas de monitoreo y trazabilidad, lo que facilita tanto el mantenimiento como la evolución de la arquitectura.

2.6.5 Patrón Reintentar (*Retry*)

Microsoft (2025), lo describe como un patrón de diseño orientado a incrementar la resiliencia y la capacidad de manejo de errores en sistemas distribuidos o en entornos que dependen de la comunicación con servicios externos. Su aplicación resulta pertinente cuando una operación no logra completarse de manera exitosa en el primer intento debido a fallas temporales, tales como interrupciones de red, indisponibilidad momentánea de un servicio externo u otros errores transitorios.

La idea central consiste en ejecutar nuevamente la operación fallida en múltiples ocasiones, estableciendo intervalos de tiempo entre cada intento, antes de considerarla definitivamente fallida y proceder a notificar al usuario o al sistema. Esta estrategia contribuye a mitigar errores transitorios y a mejorar la experiencia del usuario al otorgar al sistema la posibilidad de recuperarse de manera autónoma frente a circunstancias pasajeras.

[25] Backend: parte del sistema (lado del servidor) que se encarga de la lógica de negocio, almacenamiento y procesamiento de datos, y comunicación mediante APIs.

[26] Mecanismos de cacheo: los mecanismos de cacheo en un API Gateway son procedimientos mediante los cuales el gateway almacena temporalmente las respuestas de determinadas solicitudes para servirlas posteriormente sin necesidad de reenviar la petición al microservicio de origen. Este proceso permite reducir la latencia, disminuir la carga sobre los servicios backend y optimizar el rendimiento global de la arquitectura.

Aspectos importantes a considerar en su implementación:

- Definir el número máximo de intentos permitidos.
- Establecer intervalos de tiempo entre ejecuciones (estáticos o dinámicos).
- Incorporar mecanismos que impidan reintentos indefinidos y posibles bucles infinitos.
- Registrar y monitorear los intentos con el fin de facilitar análisis y ajustes posteriores.

En síntesis, se trata de un recurso efectivo para fortalecer la robustez de los sistemas distribuidos, al gestionar fallas transitorias de manera automática y controlada.

2.6.6 Patrón Corta Circuitos (*Circuit Breaker*)

Nygard (2017),[27] describe esta estrategia como un recurso fundamental para proteger sistemas distribuidos frente a fallos en cascada y garantizar una mayor resiliencia operativa. Inspirada en el funcionamiento de los disyuntores eléctricos, interrumpe automáticamente el flujo de operaciones cuando se detecta una sobrecarga, actuando como un mecanismo de defensa entre una aplicación y un servicio potencialmente inestable o degradado.

En arquitecturas modernas, especialmente en microservicios y sistemas orientados a servicios (SOA) (capítulo 1.4.1), los servicios dependen frecuentemente de otros servicios externos o internos. Si uno de estos servicios comienza a fallar —por lentitud, errores de red o sobrecarga—, las solicitudes adicionales no solo agravan la situación del servicio afectado, sino que también comprometen la estabilidad del sistema llamador. Esta estrategia previene este comportamiento al bloquear temporalmente las llamadas al servicio en falla, lo que permite que el sistema degrade su funcionalidad de manera controlada en lugar de colapsar completamente. Además, mejora la observabilidad al ofrecer métricas precisas sobre la frecuencia de fallos, los tiempos de respuesta y la estabilidad general de la arquitectura.

El circuito puede encontrarse en tres estados:

- Cerrado: las llamadas fluyen normalmente hacia el servicio. Si se detecta un número excesivo de fallos en un período determinado, el circuito cambia al estado abierto.
- Abierto: las llamadas al servicio fallan de inmediato, evitando la sobrecarga. Tras un tiempo de espera, el circuito pasa a un estado de prueba.
- Semi abierto: se permite el paso de un número limitado de llamadas para verificar la recuperación del servicio. Si las respuestas son exitosas, el circuito retorna al estado cerrado; en caso contrario, permanece abierto.

[27] Nygard es un arquitecto de software con amplia experiencia en el diseño de sistemas resilientes y listos para producción. Es autor de Release It!: Design and Deploy Production-Ready Software (2ª ed., 2017), obra en la que presenta estrategias prácticas para construir aplicaciones capaces de resistir las exigencias del entorno productivo.

Tras haber revisado los fundamentos tecnológicos que sustentan la migración hacia microservicios, resulta necesario analizar cómo interactúan y se relacionan los distintos componentes dentro de esta arquitectura. El próximo capítulo se centra en aspectos clave del diseño, como la comunicación entre servicios, la gestión de transacciones distribuidas, la cohesión y el acoplamiento.

Capítulo III – Principios de diseño y dinámica estructural de los microservicios

La arquitectura de microservicios (capítulo 1.4), redefine el diseño de los sistemas al distribuir la funcionalidad en componentes autónomos que colaboran entre sí. Este enfoque favorece la escalabilidad y la flexibilidad, aunque introduce nuevos retos en torno a la comunicación, las transacciones distribuidas y el diseño estructural.

La comunicación entre servicios es un elemento esencial, ya sea mediante interacciones síncronas o asíncronas, y debe garantizar eficiencia, seguridad y confiabilidad. Del mismo modo, la gestión de transacciones en entornos distribuidos exige mecanismos que preserven la consistencia sin recurrir a un control centralizado, recurriendo a patrones como sagas o compensaciones.

A su vez, los principios de cohesión y bajo acoplamiento orientan el diseño de los microservicios. La cohesión asegura que cada servicio se enfoque en una responsabilidad específica, mientras que el bajo acoplamiento facilita la evolución independiente de cada componente.

3.1 Comunicación en una arquitectura de microservicios

De acuerdo con Microsoft (2022), una aplicación basada en microservicios (capítulo 1.4.1), constituye un sistema distribuido que se ejecuta a través de múltiples procesos o servicios, los cuales, en la mayoría de los casos, se despliegan en diferentes servidores o *hosts*. Cada instancia de servicio se implementa habitualmente como un proceso independiente. En consecuencia, los servicios deben establecer su interacción mediante protocolos de comunicación entre procesos, tales como HTTP, AMQP o protocolos binarios como TCP, seleccionados en función de las características y necesidades específicas de cada servicio.

3.1.1 Comunicación sincrónica

Es un tipo de interacción donde los procesos o sistemas que participan deben operar en un tiempo coordinado y sincronizado. En este modelo, el emisor de un mensaje espera a que el receptor confirme la recepción y procesamiento antes de continuar con otras tareas.

Características:

- Bloqueo: el emisor puede bloquearse o detenerse temporalmente hasta que reciba una respuesta del receptor.
- Sincronización temporal: existe una dependencia temporal directa entre el emisor y el receptor.
- Secuencialidad: las acciones se realizan en un orden determinado y predecible.

3.1.2 Comunicación asincrónica

Es un tipo de interacción donde los procesos o sistemas pueden operar de manera independiente en términos de tiempo y ritmo. En este modelo, el emisor envía un mensaje al receptor y continúa con otras tareas sin esperar una respuesta inmediata. La comunicación asincrónica no requiere una sincronización temporal estricta y puede basarse en mecanismos como colas de mensajes o eventos para gestionar la entrega y recepción de datos.

Características:

- No bloqueante: el emisor no se detiene ni bloquea mientras espera una respuesta del receptor.
- Independencia temporal: no hay una dependencia temporal directa entre el emisor y el receptor.
- Eventos o colas: se pueden utilizar eventos, *callbacks* o colas de mensajes para manejar la recepción y el procesamiento de mensajes.

3.1.3 Tipos de comunicación

Según Microsoft (2022), en arquitecturas basadas en microservicios la comunicación entre clientes y servicios puede clasificarse a partir de dos ejes fundamentales: el tipo de protocolo y la cantidad de receptores.

El primer eje distingue entre protocolos sincrónicos y asincrónicos (capítulo 3.1.1 y capítulo 3.1.2). Los protocolos sincrónicos, como HTTP, requieren que el cliente envíe una solicitud y espere una respuesta antes de continuar con su ejecución. Aunque la programación del cliente puede desarrollarse de manera sincrónica o asincrónica, la naturaleza del protocolo permanece sincrónica, lo que restringe el avance del flujo de trabajo hasta que se recibe una respuesta. En contraste, los protocolos asincrónicos permiten que los mensajes se transmitan sin necesidad de obtener una respuesta inmediata, favoreciendo una comunicación desacoplada entre los servicios.

El segundo eje considera la cantidad de receptores. En las comunicaciones de receptor único, cada mensaje es atendido por un servicio específico. En cambio, en las comunicaciones con múltiples receptores, una publicación puede ser procesada por varios servicios o incluso quedar sin atender. Para este último caso suelen emplearse mecanismos asincrónicos basados en patrones de publicación y suscripción, característicos de arquitecturas orientadas a eventos. Dichos mecanismos se implementan, por lo general, mediante buses de eventos o *brokers* de mensajería.

En la práctica, una aplicación basada en microservicios tiende a combinar ambos tipos de comunicación. Predomina el uso de protocolos sincrónicos para interacciones directas a través de HTTP/HTTPS, complementados con protocolos de mensajería que permiten la coordinación asincrónica entre servicios. No obstante, el principal desafío en

el diseño de este tipo de arquitecturas consiste en lograr una integración asincrónica que preserve tanto la independencia como la escalabilidad de los servicios.

3.2 Transacciones

Marteens (1998), sostiene que una transacción consiste en una secuencia de operaciones de lectura y escritura, durante las cuales la base de datos se percibe como un conjunto consistente y, en caso de producirse actualizaciones, debe garantizarse que permanezca en un estado igualmente consistente. En la misma línea, Date (2001), define una transacción como una unidad lógica de trabajo que, por lo general, comprende múltiples operaciones sobre una base de datos, en particular diversas operaciones de actualización.

Por su parte, Newman (2021), amplía la definición señalando que, en términos generales, una transacción en el ámbito de la computación representa un conjunto de una o más acciones que deben ser tratadas como una única unidad. Así, cuando se realizan múltiples modificaciones en el marco de una operación global, el resultado se confirma únicamente si todos los cambios se completan de manera correcta; de lo contrario, el proceso se revierte. En el contexto de las bases de datos, las transacciones aseguran que las modificaciones de estado —como eliminaciones, inserciones o actualizaciones— se ejecuten adecuadamente, pudiendo abarcar una o varias tablas dentro de una misma operación transaccional.

3.2.1 Propiedades ACID

El acrónimo ACID describe las propiedades claves de las transacciones (*atomicity, consistency, isolation y durability*), en español, atomicidad, consistencia, aislamiento y durabilidad.

- Atomicidad: asegura que las operaciones realizadas dentro de la transacción, se completen todas o ninguna. Si algún cambio de los que se está intentando realizar falla por alguna razón, se aborta toda la operación y es como si nunca se hubieran realizado cambios.
- Consistencia: cuando se realizan cambios en la base de datos, se asegura que la base de datos quede en un estado válido y consistente.
- Aislamiento: permite que múltiples transacciones operen al mismo tiempo, sin interferir. Esto se logra asegurando que cualquier cambio de estado intermedio realizado durante una transacción sea invisible para otras transacciones.
- Durabilidad: se refiere a que cuando una transacción se confirma, los cambios sólo pueden deshacerse mediante otra transacción.

3.2.2 Transacciones distribuidas

Una transacción distribuida involucra recursos dispersos en distintos sistemas o bases de datos, los cuales pueden encontrarse en diferentes entornos físicos y bajo administraciones heterogéneas. Este tipo de transacción requiere la coordinación de múltiples gestores transaccionales pertenecientes a los sistemas distribuidos, apoyándose en un coordinador principal encargado de supervisar los procesos de confirmación (*commit*) o reversión (*rollback*) en los sistemas participantes, con el propósito de garantizar que todos los cambios se mantengan consistentes y seguros.

Este enfoque incrementa la complejidad del sistema debido a la necesidad de sincronizar operaciones y asegurar la coherencia de los datos en múltiples entornos. Además, puede generar mayor latencia e implicar la utilización de protocolos avanzados que gestionen de manera adecuada tanto la confirmación de las operaciones como el aislamiento entre transacciones concurrentes.

En el contexto de una arquitectura de microservicios (capítulo 1.4), estas dificultades se intensifican, dado que cada servicio administra su propia base de datos y opera de manera autónoma. La coordinación de una transacción que abarque varios servicios implica no solo la gestión técnica de los recursos distribuidos, sino también la preservación de la consistencia en procesos de negocio que atraviesan múltiples dominios. Por esta razón, en lugar de depender exclusivamente de transacciones distribuidas tradicionales, se utilizan otras técnicas que permiten mantener la coherencia sin comprometer la autonomía de los microservicios ni afectar su capacidad de escalar de manera independiente.

3.2.3 Sagas

A diferencia de los compromisos de dos fases[28], una saga constituye, por diseño, un algoritmo capaz de coordinar múltiples cambios de estado evitando la necesidad de mantener recursos bloqueados durante períodos prolongados. Este mecanismo se fundamenta en la descomposición del proceso en pasos discretos que pueden ejecutarse de manera independiente.

Según Garcia-Molina & Salem (1987), en el artículo Sagas se expone la idea central de gestionar de manera más eficiente las denominadas transacciones de larga duración (*Long-Lived Transactions,* LLTs). Estas transacciones pueden extenderse durante minutos, horas o incluso días y, en ese lapso, requieren realizar modificaciones sobre una base de datos.

[28] Al-Houmaily & Samaras (2000), el protocolo de compromiso en dos fases (2PC), es un protocolo de sincronización utilizado para garantizar la atomicidad global de las transacciones en sistemas distribuidos

Si una LLT se modela como una transacción convencional, esta abarca todo su ciclo de vida, ocasionando que múltiples filas o incluso tablas completas permanezcan bloqueadas durante un tiempo considerable. Esto genera problemas significativos al intentar otros procesos acceder a dichos recursos. Para resolver esta situación, el artículo propone descomponer las LLTs en una secuencia de transacciones independientes, cada una de corta duración y con impacto limitado sobre los datos. Este enfoque reduce la contención en la base de datos y optimiza la gestión de los bloqueos.

Aunque inicialmente concebidas para escenarios con una única base de datos, las sagas resultan igualmente efectivas en arquitecturas distribuidas. En este contexto, un proceso de negocio puede dividirse en un conjunto de interacciones con múltiples servicios colaboradores, conformando en su conjunto una saga.

Es importante señalar que una saga no garantiza atomicidad en el sentido estricto de las propiedades ACID de una transacción convencional (capítulo 3.2.1). La atomicidad se preserva únicamente en el ámbito de cada transacción individual, no en el de la saga completa. No obstante, la saga proporciona información suficiente para comprender el estado del proceso, delegando en los desarrolladores la gestión de sus implicaciones.

Modos de fallas de la saga

Al estar compuesta por transacciones individuales, una saga requiere definir mecanismos de recuperación ante fallos. Garcia-Molina & Salem (1987), distinguen dos modalidades principales: recuperación hacia atrás (*backward recovery*) y recuperación hacia adelante (*forward recovery*).

La primera consiste en revertir los efectos de una transacción fallida mediante operaciones compensatorias, lo cual equivale a un *rollback*. La segunda implica reanudar el proceso desde el punto en que se produjo el fallo, repitiendo las transacciones necesarias. Para ello, resulta indispensable conservar información suficiente que permita el reintento. En función del proceso de negocio, la estrategia de recuperación puede basarse en una u otra modalidad, o bien en una combinación de ambas. Cabe destacar que las sagas permiten recuperarse de fallos de negocio, pero no de fallos técnicos.

Implementación de sagas

Existen dos enfoques predominantes para la implementación de sagas: la coreografía y la orquestación.

En la coreografía, los participantes coordinan sus acciones mediante el intercambio de eventos, sin un controlador centralizado. Cada transacción local publica eventos de dominio que desencadenan la ejecución de otras transacciones en servicios distintos.

Por su parte, la orquestación introduce un orquestador central que supervisa el flujo de la saga. Este componente indica a cada participante qué transacción ejecutar,

gestiona los estados intermedios y coordina la recuperación de errores mediante transacciones de compensación.

3.3 Cohesión

DeMarco (1979), definió la cohesión como la medida en que los componentes de un módulo están relacionados entre sí. En general, cuanto más estrechamente relacionadas estén las tareas de un módulo, mayor será su cohesión.

Constantine & Yourdon (1979), definieron la cohesión como la medida en que los elementos internos de un módulo trabajan juntos para lograr un propósito común.

En arquitecturas basadas en microservicios, la cohesión resulta crítica, ya que cada servicio debe tener una responsabilidad bien delimitada. De esta manera, se asegura que cada microservicio sea autónomo, mantenible y escalable, evitando la dispersión de funciones dentro de un mismo componente.

3.3.1 Niveles de cohesión

Constantine & Yourdon (1979), identifican siete niveles de cohesión, cada uno definido por un principio asociativo específico que determina la fuerza de la cohesión. A continuación se detallan los mismos.

- Cohesión Coincidental: los elementos de un módulo están agrupados arbitrariamente, sin una relación significativa entre ellos. La cohesión coincidental básicamente, establece un cero en la escala o jerarquía de cohesión.
- Cohesión Lógica: los elementos de un módulo están asociados de manera lógica, es decir, pertenecen a la misma clase lógica de funciones similares o relacionadas. La cohesión lógica es generalmente más fuerte y deseable que la cohesión coincidental.
- Cohesión Temporal: los elementos están agrupados porque se ejecutan en el mismo período de tiempo. La cohesión temporal implica que todas las ocurrencias de todos los elementos de procesamiento en una colección ocurren dentro del mismo período limitado de tiempo durante la ejecución del sistema.
- Cohesión Procedural: los elementos están agrupados porque están involucrados en la ejecución de una secuencia específica de pasos, dentro de un algoritmo o función.
- Cohesión Comunicacional: la cohesión comunicacional es el nivel más bajo en el cual se encuentra una relación entre elementos de procesamiento que son intrínsecamente dependiente del problema. Cuando un conjunto de elementos de procesamiento está asociado comunicacionalmente significa que todos los elementos operan sobre el mismo conjunto de datos de entrada y/o producen el mismo conjunto de datos de salida.

- Cohesión Secuencial: los datos de salida (o resultados) de un elemento de procesamiento sirven como datos de entrada para el siguiente elemento de procesamiento.
- Cohesión Funcional: en un módulo completamente funcional, cada elemento de procesamiento es una parte integral y esencial para el desempeño de una única función.

3.4 Acoplamiento

Myers (1978), definió el acoplamiento como la medida en que un módulo depende de otros módulos.

Page-Jones (1980), consideró el acoplamiento como la medida en que un módulo utiliza los servicios de otro módulo.

Schach (1991), describió el acoplamiento como la medida en que los componentes de un sistema dependen unos de otros. Un sistema está bien acoplado si los cambios en un componente tienen un impacto mínimo en otros componentes.

En arquitecturas basadas en microservicios el bajo acoplamiento es esencial, ya que permite que los servicios evolucionen, se desplieguen y escalen de manera independiente, sin afectar al resto del sistema.

3.4.1 Tipos de acoplamiento

Constantine & Yourdon (1979), definen seis tipos de acoplamientos. A continuación se detallan los mismos.

- Acoplamiento de Datos: los módulos están acoplados si comparten datos a través de parámetros. Esto significa que los módulos solo se comunican intercambiando datos, sin compartir lógica interna.
- Acoplamiento de Control: los módulos están acoplados si un módulo controla el flujo de otro módulo mediante la transferencia de información de control. Esto implica que un módulo toma decisiones sobre la ejecución de otro módulo.

  Acoplamiento de Estampado: los módulos están acoplados si comparten una estructura de datos compleja, como una estructura de registro o una matriz multidimensional. Cambios en la estructura de datos pueden requerir modificaciones en múltiples módulos.
- Acoplamiento de Contenido: los módulos están acoplados si comparten información interna, como variables globales o accesos directos a estructuras de datos. Esto implica una dependencia directa de la implementación interna de otro módulo.
- Acoplamiento Común: los módulos están acoplados si comparten datos globales. Esto significa que varios módulos tienen acceso a los mismos datos globales, lo que puede dificultar el mantenimiento y la comprensión del código.

- Acoplamiento de Mensajes: los módulos están acoplados si se comunican a través de mensajes, es decir, intercambian información mediante la invocación de métodos o funciones, pero sin compartir datos directamente.

3.5 Relación entre acoplamiento y cohesión

Myers (1978), señaló que la cohesión y el acoplamiento son dos objetivos opuestos en el diseño de sistemas de *software*. Argumentó que mientras mayor sea la cohesión dentro de un módulo, es decir, más fuerte sea la relación entre las partes del módulo, menor será el acoplamiento deseable entre los módulos. Esta relación inversa implica que mejorar la cohesión generalmente requiere reducir el acoplamiento y viceversa.

DeMarco (1979), también enfatizó la importancia de acoplamiento y cohesión. Sugirió que un buen diseño debería tener alta cohesión dentro de los módulos, lo que significa que las funciones y responsabilidades de un módulo deben estar estrechamente relacionadas entre sí. Además, abogó por un bajo acoplamiento entre los módulos para minimizar la dependencia entre ellos y facilitar cambios futuros en el sistema.

El estudio de la comunicación, las transacciones, la cohesión y el acoplamiento permite comprender los retos inherentes a los sistemas distribuidos. Con estos conceptos como punto de partida, el capítulo siguiente introduce enfoques prácticos para formular microservicios a partir de sistemas monolíticos, haciendo hincapié en metodologías como el diseño dirigido por el dominio y en distintos modelos de migración.

Capítulo IV – Estrategias y enfoques para la transición del monolítico a microservicios

El presente capítulo aborda los fundamentos conceptuales y prácticos necesarios para la transición de un sistema monolítico hacia una arquitectura basada en microservicios. En primer lugar, se introduce el Diseño Dirigido por el Dominio (DDD) como enfoque metodológico que permite estructurar el sistema a partir de la lógica del negocio y de sus distintos dominios. Este paradigma constituye una base esencial para garantizar la coherencia y la alineación entre los requerimientos organizacionales y la implementación técnica.

Posteriormente, se analizan los enfoques de migración de sistemas monolíticos a microservicios, entre los cuales se incluyen el *Big Bang*, el enfoque incremental conocido como *Strangler Fig* y la propuesta de Richardson y Smith. La exposición de estas alternativas permite comprender los riesgos, ventajas y limitaciones inherentes a cada estrategia, así como su aplicabilidad en distintos contextos.

En una tercera instancia, se profundiza en el concepto de Prueba de Concepto (PoC) como mecanismo de validación previa a la adopción de microservicios. Se examina su definición, su importancia en el ámbito del desarrollo de software, los componentes esenciales que la conforman y los beneficios que aporta al proceso de transición tecnológica.

Finalmente, se aborda la relevancia de los equipos interdisciplinarios en este tipo de proyectos. La participación de profesionales con diversas competencias técnicas y de negocio resulta indispensable para garantizar que la transformación arquitectónica no solo responda a criterios tecnológicos, sino también a objetivos estratégicos de la organización.

4.1 Diseño dirigido por el dominio (DDD)

Propuesto por Evans (2003), constituye una metodología de desarrollo de *software* orientada a gestionar la complejidad inherente de los sistemas con una elevada carga de lógica de negocio. A diferencia de los enfoques centrados exclusivamente en los aspectos técnicos, DDD enfatiza la construcción de un modelo de dominio, entendido como una representación conceptual precisa del problema que se busca resolver. Según Evans, la solidez y flexibilidad de un sistema dependen de la colaboración estrecha entre expertos del dominio y desarrolladores, así como de la constante alineación del *software* con el conocimiento del negocio.

Uno de los pilares esenciales de DDD es el uso de un lenguaje ubicuo, un vocabulario común y coherente que comparten tanto los equipos técnicos como los no técnicos, y que se refleja de manera directa en el código del sistema. Este lenguaje compartido favorece la comunicación y reduce ambigüedades en el desarrollo. Asimismo,

el enfoque promueve la creación de modelos conceptuales estrechamente vinculados al dominio, los cuales evolucionan de forma paralela al conocimiento del negocio.

DDD introduce además la noción de contextos delimitados (*bounded contexts*), que permiten dividir el sistema en áreas bien definidas con modelos independientes. Este mecanismo facilita la modularidad y la integración de subdominios múltiples sin comprometer la claridad conceptual. Dentro de cada contexto delimitado, se aplican diversos patrones tácticos —como entidades, objetos de valor, agregados, repositorios y servicios— que contribuyen a implementar el modelo de dominio de manera coherente, mantenible y alineada con los objetivos del negocio.

4.2 Enfoques para formular microservicios a partir de un sistema monolítico

4.2.1 Enfoque del Big Bang

Fowler (2014), la teoría del "*Big Bang*" en el contexto de la migración de sistemas monolíticos a microservicios se refiere a una estrategia drástica y riesgosa donde se intenta reescribir o rediseñar completamente un sistema monolítico como un conjunto de microservicios en un solo movimiento o "gran explosión". Martin Fowler, un reconocido experto en arquitectura de *software*, menciona esta estrategia como una opción, pero generalmente no la recomienda debido a sus altos riesgos y dificultades. A continuación una descripción del enfoque *Big Bang:*

1. Rediseño completo: se planea y se implementa una reescritura completa del sistema existente. Esto significa que el equipo de desarrollo se embarca en un proyecto de rediseño de todos los componentes y funcionalidades del sistema monolítico en términos de microservicios, desde el principio.
2. Implementación paralela: durante el desarrollo del nuevo sistema basado en microservicios, el sistema monolítico existente sigue operando. El nuevo sistema se desarrolla en paralelo sin afectar el funcionamiento del sistema actual hasta el momento del despliegue.
3. Despliegue único: una vez que el nuevo sistema basado en microservicios está completo, se realiza un despliegue "*Big Bang*" donde se apaga el sistema monolítico y se reemplaza completamente por el nuevo sistema. Este despliegue se realiza en un único evento de gran escala.

Este enfoque es atractivo por la posibilidad de obtener, de manera inmediata, todos los beneficios asociados con la arquitectura de microservicios, como una mayor escalabilidad, mantenibilidad y estandarización tecnológica desde el inicio del nuevo diseño. Sin embargo, esta aparente ventaja viene acompañada de una carga significativa de riesgo y complejidad. El despliegue único implica que cualquier error o imprevisto puede comprometer seriamente la operación del sistema en su totalidad. La magnitud del cambio, al concentrarse en un solo evento, aumenta las probabilidades de fallos y

requiere un nivel de planificación y ejecución extremadamente riguroso. Además, el proceso demanda un esfuerzo de tiempo, recursos y coordinación, lo que puede resultar costoso tanto para el equipo técnico como para la organización. Por lo tanto, aunque la promesa de una transición total y rápida puede parecer atractiva, el enfoque *Big Bang* conlleva desafíos sustanciales que lo convierten en una opción poco recomendada para la mayoría de los casos.

4.2.2 Enfoque incremental Strangler Fig

Según Fowler (2014), es un enfoque gradual y sistemático para migrar aplicaciones monolíticas hacia una arquitectura de microservicios (capítulo 1.3 y capítulo 1.4). La estrategia plantea sustituir progresivamente las funcionalidades del monolítico mediante la incorporación de microservicios, reduciendo con ello los riesgos asociados a una transformación abrupta y garantizando la continuidad operativa del sistema.

El proceso inicia con un análisis exhaustivo del sistema monolítico, cuyo propósito es identificar módulos y componentes que puedan ser separados sin comprometer la funcionalidad global. En esta fase, las técnicas de diseño dirigido por el dominio, (capítulo 4.1), suelen resultar particularmente útiles, ya que permiten definir límites claros de contexto (*bounded contexts*) que sirven como base para diseñar microservicios independientes.

Posteriormente, el desarrollo de nuevos componentes se orienta hacia la arquitectura de microservicios desde el inicio. Así, toda funcionalidad que se incorpore al sistema se implementa como un servicio independiente, evitando incrementar la complejidad del monolítico. De forma paralela, se seleccionan módulos ya existentes que pueden ser refactorizados y trasladados al nuevo esquema. Este proceso, llevado a cabo de manera paulatina, facilita una migración controlada, módulo por módulo.

Un aspecto fundamental de esta estrategia consiste en la gestión de las solicitudes. Para ello, se introduce un enrutador o *proxy* inverso encargado de interceptar y redirigir las peticiones entrantes. Las solicitudes correspondientes a funcionalidades migradas se dirigen hacia los microservicios, mientras que aquellas aún no transformadas continúan siendo gestionadas por el monolítico. Este mecanismo asegura la coexistencia ordenada de ambas arquitecturas durante el proceso de transición.

El desacoplamiento y la extracción de datos constituyen un desafío fundamental. A medida que se trasladan funcionalidades, resulta necesario migrar también los datos asociados, lo que puede implicar la replicación de información, la sincronización de bases de datos y la implementación de mecanismos que garanticen la consistencia entre el monolítico y los microservicios. En este sentido, el diseño de interfaces de servicio claras y bien estructuradas es esencial para asegurar una comunicación eficiente y desacoplada entre los distintos componentes.

Asimismo, la estrategia requiere de un monitoreo y evaluación continua. Cada microservicio incorporado debe ser sometido a pruebas rigurosas y a mecanismos de observabilidad que garanticen su correcto funcionamiento y su integración con el resto del sistema. Del mismo modo, resulta indispensable evaluar periódicamente el rendimiento y la fiabilidad de los servicios, realizando los ajustes necesarios para optimizar su operación.

La fase final del proceso consiste en la reducción progresiva del monolítico. A medida que las funcionalidades críticas se trasladan a microservicios, el sistema monolítico va perdiendo relevancia y puede ser desactivado gradualmente.

En términos de beneficios, la estrategia *Strangler Fig* destaca por su capacidad de minimizar los riesgos inherentes a una migración, ya que permite introducir cambios de manera controlada y con escasa interrupción del servicio. Los usuarios pueden seguir utilizando la aplicación mientras se lleva a cabo la transformación, lo que asegura la continuidad del negocio. Además, este enfoque ofrece flexibilidad y adaptabilidad frente a cambios en los requisitos tecnológicos o del mercado, y facilita la implementación de mejoras continuas sin necesidad de acometer reestructuraciones masivas.

No obstante, esta metodología también plantea importantes desafíos. La coexistencia prolongada de un sistema monolítico junto con microservicios incrementa la complejidad del entorno, lo que exige una planificación cuidadosa y una gestión eficiente. La sincronización de datos entre ambas arquitecturas constituye, además, un problema técnico de gran envergadura que puede requerir soluciones avanzadas. Por último, la operación en paralelo genera una sobrecarga inicial en términos de infraestructura y administración, ya que el sistema debe soportar simultáneamente dos arquitecturas hasta que el proceso de migración se complete en su totalidad.

4.2.3 Enfoque propuesto por Richarson y Smith

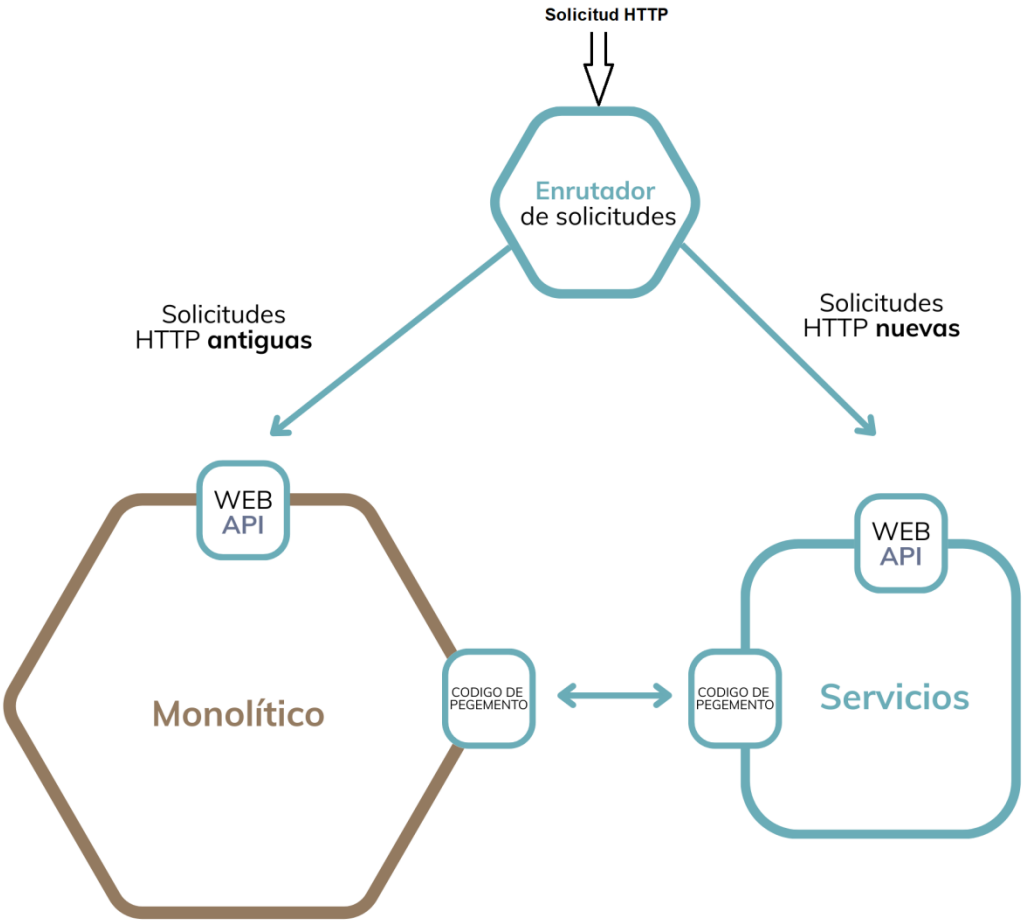
Richarson & Smith (2016), interpretan la transformación de sistemas monolíticos hacia arquitecturas de microservicios como un proceso de modernización de *software* que se apoya en prácticas históricas de refactorización. Ambos autores advierten sobre los peligros del enfoque *Big Bang*, señalando que la reescritura completa de un sistema suele derivar en proyectos fallidos debido a sus altos costos y riesgos. En contraposición, proponen una estrategia incremental que combina la refactorización del monolítico con la incorporación progresiva de microservicios, garantizando que ambas arquitecturas coexistan de manera controlada durante el proceso de transición.

Este modelo se articula en torno a tres estrategias principales. La primera, denominada “dejar de excavar” (*stop digging*), establece que el monolítico no debe seguir creciendo: toda nueva funcionalidad debe desarrollarse directamente como microservicio independiente. Esta decisión previene que el sistema monolítico se vuelva más complejo

y permite que los beneficios de los microservicios, como escalabilidad y despliegues independientes, se experimenten desde el inicio. La Figura 11 muestra la arquitectura del sistema después de aplicar esta estrategia

Figura 11

Nueva funcionalidad implementada como servicio



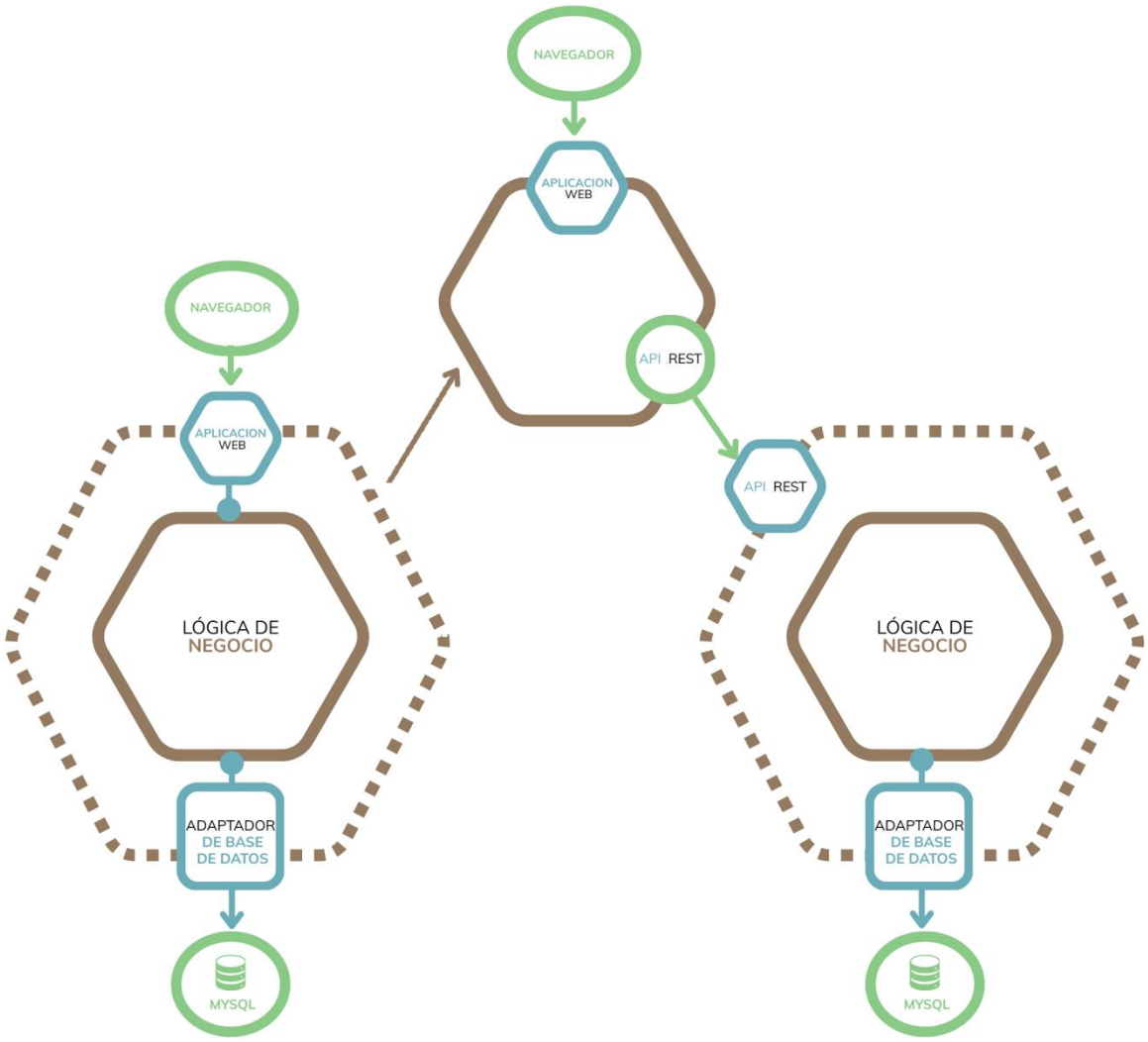
Nota. Adaptada de Richardson & Smith (2016). *Microservices from design to deployment*. NGINX

La segunda estrategia, “separar el *frontend* y el *backend”* (*split frontend and backend*), plantea dividir la capa de presentación de las capas de negocio y de acceso a datos. Esta separación permite desarrollar y escalar ambas partes de manera independiente, lo cual resulta particularmente ventajoso para acelerar la evolución de la interfaz de usuario, realizar pruebas y exponer APIs reutilizables para otros servicios. Sin embargo, los autores advierten que esta solución no resuelve todos los problemas, ya que uno o ambos subsistemas resultantes podrían seguir siendo monolíticos y, por ende,

difíciles de gestionar. La Figura 12 muestra la arquitectura antes y después de la refactorización.

Figura 12

Refactorizando un aplicación existente



Nota. Adaptada de Richardson & Smith (2016). *Microservices from design to deployment*. NGINX

La tercera estrategia, “extraer servicios” (*extract services*), consiste en identificar módulos dentro del monolítico y convertirlos en microservicios. La recomendación es iniciar con módulos fáciles de extraer para ganar experiencia, y posteriormente abordar aquellos que aporten mayor beneficio, como los que cambian con frecuencia o requieren

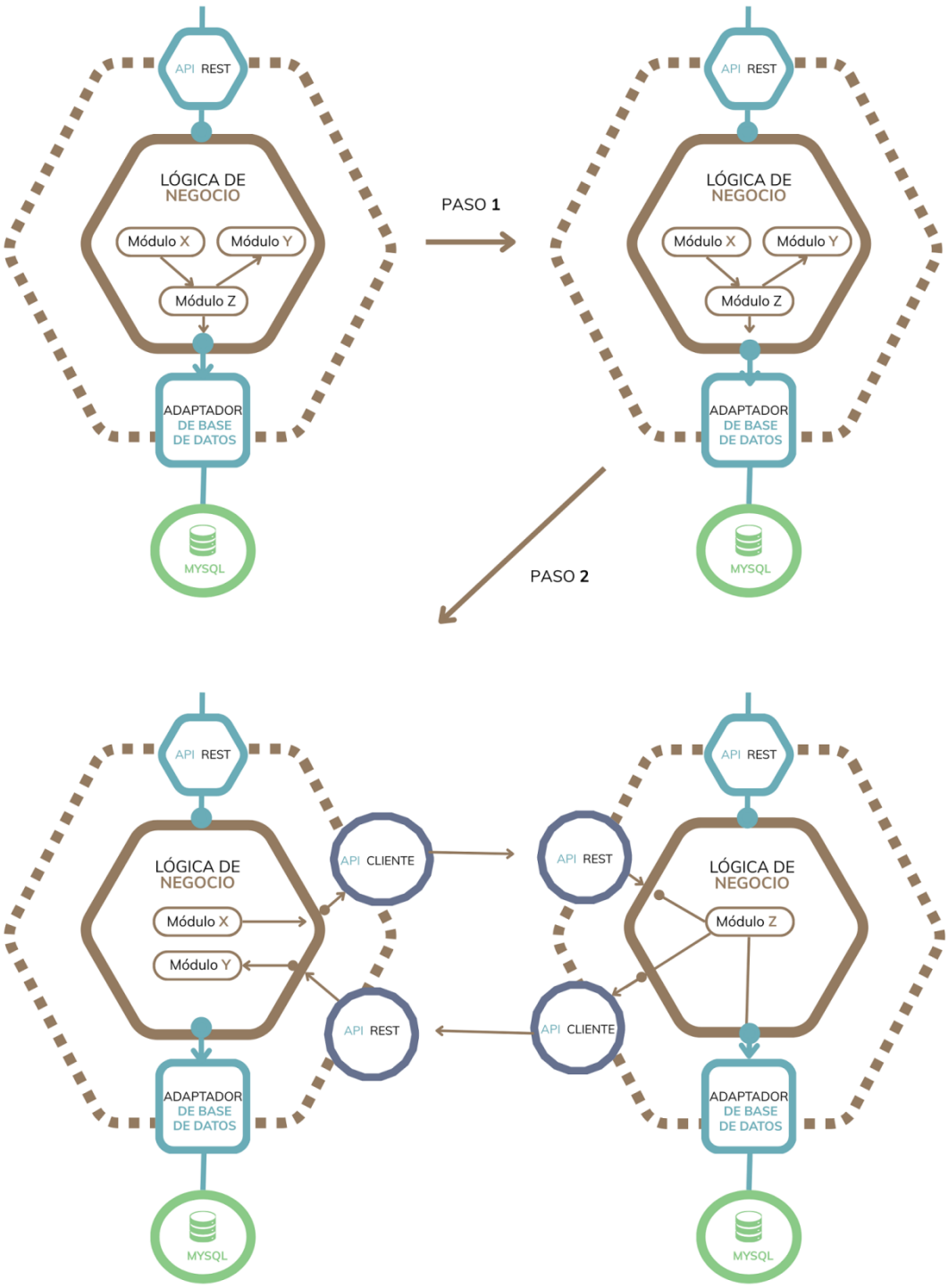
alta capacidad computacional. Este proceso implica definir interfaces de grano grueso[29] y, en ocasiones, introducir una capa anticorrupción que actúe como intermediaria entre el modelo de dominio heredado y el nuevo servicio. Aunque el desarrollo de esta capa puede ser complejo, resulta esencial para evitar que los microservicios adopten conceptos obsoletos del monolítico. La Figura 13 muestra la refactorización del módulo Z.

Las ventajas del enfoque de Richardson y Smith radican en su carácter progresivo, la reutilización de prácticas consolidadas y la posibilidad de obtener resultados tangibles en cada etapa de la migración. Además, este modelo facilita la introducción de microservicios sin necesidad de interrumpir el funcionamiento del sistema existente. No obstante, los desafíos no son menores: el diseño y mantenimiento de las capas de integración puede demandar un esfuerzo significativo, y la gestión de datos compartidos entre el monolítico y los nuevos servicios incrementa la complejidad. En ocasiones, la refactorización necesaria para extraer módulos también requiere modificaciones profundas en el código heredado, lo que eleva los costos de implementación.

[29] Hohpe & Woolf (2003), una interfaz de grano grueso se define como aquella que expone un conjunto de funcionalidades de forma amplia dentro de una sola operación, en lugar de fragmentar el acceso en múltiples invocaciones pequeñas y específicas. Este enfoque resulta especialmente adecuado en sistemas distribuidos y arquitecturas orientadas a servicios, ya que reduce la necesidad de llamadas remotas frecuentes y, en consecuencia, minimiza la latencia y la sobrecarga de comunicación.

Figura 13

Refactorizando el módulo Z del monolítico en microservicios



Nota. Adaptada de Richardson & Smith (2016). *Microservices from design to deployment*. NGINX

4.3 Pruebas de conceptos (PoC)

4.3.1 Definición de una PoC

Según Atlassian (2025), una prueba de conceptos o en inglés *Proof of Concept* (PoC), es un proceso mediante el cual se recopilan evidencias para respaldar la factibilidad de un proyecto. Los gestores de proyectos la implementan en las fases tempranas del desarrollo para demostrar la viabilidad del proyecto a equipos de producto, clientes y otras partes interesadas. Este proceso puede revelar fallos que lleven a revisar o abandonar un proyecto, o confirmar su probabilidad de éxito, proporcionando pruebas de su viabilidad para el desarrollo.

4.3.2 Aplicación de las PoC en el desarrollo del software

Según GeeksforGeeks (2025), una PoC permite probar la viabilidad de una idea de software a pequeña escala. Su objetivo es demostrar que la solución puede construirse, que funciona en la vida real, que puede resolver problemas existentes y que puede generar beneficios. La PoC puede realizarse en cualquier punto del ciclo de vida del desarrollo de software, ya sea al inicio para probar la viabilidad de toda la idea o a mitad del proyecto para probar una característica específica. Los entregables de la demostración de concepto pueden tomar diversas formas, como documentos, presentaciones o código escrito.

4.3.3 Importancia de la PoC

Según GeeksforGeeks (2025), el desarrollo de una PoC es crucial para ahorrar tiempo, dinero y esfuerzo, evitando invertir en ideas que podrían no ser viables. Ayuda a realizar predicciones más precisas sobre los gastos y otros recursos necesarios. Validar la idea lo antes posible es el primer paso hacia una mejor adaptación al mercado y aumento de la financiación. Según Forbes[30], el 90% de las *startups*[31] fracasan por estas razones. La PoC es la base para el producto futuro.

4.3.4 Componentes clave de la PoC

Según GeeksforGeeks (2025), las componentes de una PoC son las siguientes:

- Definir la necesidad**:** identificar los problemas del mundo real que el software resolverá para validar las suposiciones iniciales.
- Establecer criterios de éxito: determinar los puntos de referencia para medir el éxito o fracaso del proyecto.
- Enumerar los recursos necesarios: crear una lista exhaustiva de recursos tangibles e intangibles necesarios para ejecutar el proyecto.

[30] Forbes (2025), es una revista especializada en el mundo de los negocios y las finanzas

[31] Blank (2013), la define como una organización temporal en busca de un modelo de negocio repetible y escalable. El término suele asociarse con empresas emergentes, particularmente de base tecnológica.

- Determinar el cronograma: crear una hoja de ruta del producto que describa el cronograma desde la ideación hasta el desarrollo.
- Desarrollar y probar el prototipo: desarrollar y probar el prototipo con el público objetivo, recopilando comentarios.
- Revisar y refinar: evaluar cómo el prototipo se desempeñó frente a los criterios de éxito predefinidos y utilizar la información para mejorar las áreas que no cumplieron con el éxito.
- Presentar la PoC: presentar la idea a las partes interesadas para su aprobación de desarrollo, proporcionando un modelo claro de cómo funciona la idea.

4.3.5 Beneficios de crear una PoC

Según GeeksforGeeks (2025), los beneficios de una PoC son las siguientes:

- Ahorro de tiempo y dinero: evita desarrollar un producto completo sin evaluar su viabilidad técnica o comercial.
- Elección de la pila tecnológica adecuada: permite validar el conjunto de herramientas que mejor se adapta a los requisitos actuales y objetivos a largo plazo.
- Evaluación de riesgos tecnológicos: identifica riesgos tecnológicos desde el principio del proceso de desarrollo.
- Confianza de las partes interesadas: reduce la posibilidad de desarrollar el producto incorrecto al obtener comentarios tempranos.
- Convencer a los inversores: una PoC tangible puede persuadir a los inversores para financiar el MVP[32]
- Base para el desarrollo futuro: los recursos invertidos en la PoC pueden utilizarse para desarrollar el MVP.

4.4 Equipos interdisciplinarios

Raeburn (2025), los equipos interdisciplinarios, también conocidos como equipos multifuncionales, están compuestos por profesionales de diversas áreas que colaboran para alcanzar objetivos comunes. Esta estructura promueve la integración de habilidades y perspectivas variadas, lo que puede conducir a soluciones más innovadoras y eficientes.

La creación de equipos interdisciplinarios efectivos implica cuatro pasos claves:

1. Fomentar la diversidad: incluir miembros de diferentes departamentos, niveles de experiencia y antecedentes culturales para enriquecer el enfoque del equipo.
2. Alinear los objetivos: establecer metas claras y compartidas que conecten el trabajo diario con los objetivos generales de la organización.

[32] MVP: Siglas para (*Minimum Viable Product*), en español, Producto Mínimo Viable.

3. Incluir expertos en la materia: contar con especialistas que aporten conocimientos técnicos y guíen al equipo en áreas específicas.
4. Adoptar la automatización del trabajo: utilizar herramientas que reduzcan tareas repetitivas y mejoren la eficiencia del equipo.

Los beneficios de los equipos interdisciplinarios incluyen una mayor innovación, mejor toma de decisiones y una ejecución más rápida de proyectos complejos. Sin embargo, también pueden enfrentar desafíos como la coordinación entre diferentes departamentos y la gestión de conflictos. Para superar estos obstáculos, es esencial establecer una comunicación clara, definir roles y responsabilidades, y utilizar herramientas de gestión de proyectos que faciliten la colaboración.

En resumen, los equipos interdisciplinarios ofrecen una estructura flexible y colaborativa que puede mejorar significativamente el rendimiento organizacional cuando se implementan con una planificación y gestión adecuadas.

Finalmente, los fundamentos presentados en este capítulo y los anteriores brindan el soporte conceptual para estructurar una propuesta metodológica propia. El capítulo siguiente expone dicha metodología, organizada en etapas y fases, que orientan de manera sistemática el proceso de rediseño de un ERP monolítico hacia una arquitectura de microservicios.

Capítulo V – Metodología propuesta para el rediseño de un ERP legacy monolítico hacia microservicios

El presente capítulo tiene como propósito exponer la metodología de alto nivel propuesta para abordar el proceso de transición de un sistema ERP *legacy* monolítico hacia una arquitectura de microservicios. A diferencia de los capítulos anteriores, en los cuales se desarrollaron los conceptos teóricos, las tecnologías habilitadoras y los principios de diseño que sustentan la transformación, en esta sección se plantea una estrategia metodológica que integra dichos aportes en un esquema práctico y aplicable.

La estructura del capítulo contempla tres etapas: el diagnóstico inicial del sistema, la realización de pruebas de concepto orientadas a reducir incertidumbre y, finalmente, el diseño incremental apoyado en el enfoque *Strangler Fig* y en diversos criterios de extracción de módulos. Cada una de estas etapas se encuentra directamente vinculada con los fundamentos teóricos presentados en los capítulos 1, 2, 3 y 4, garantizando que la propuesta metodológica se encuentre sólidamente sustentada tanto en la literatura académica como en las buenas prácticas de la industria del *software*.

De esta manera, el capítulo constituye la síntesis metodológica de la tesina, ofreciendo una hoja de ruta gradual, controlada y fundamentada para guiar el rediseño de sistemas ERP monolíticos hacia arquitecturas de microservicios.

5.1 Definición conceptual de la metodología

La metodología propuesta se concibe como un marco de alto nivel diseñado para guiar el proceso de rediseño de un sistema ERP l*egacy*, desarrollado bajo una arquitectura monolítica, hacia un modelo basado en microservicios. Esta metodología constituye una hoja de ruta conceptual que integra fundamentos teóricos, enfoques de descomposición y prácticas de validación progresiva.

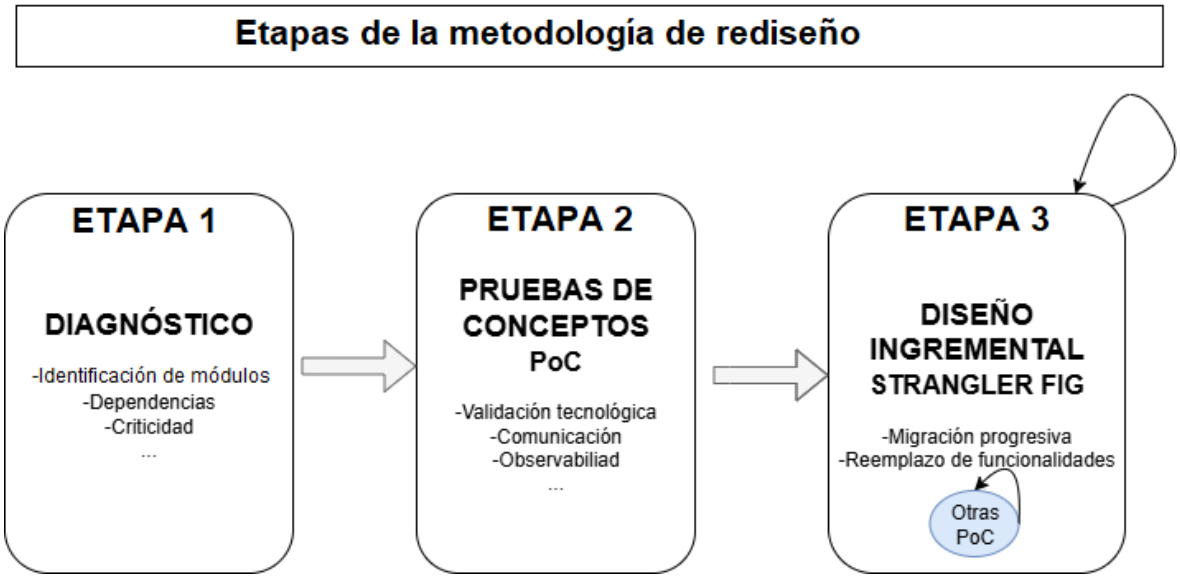
En términos metodológicos, la propuesta se sustenta en tres ejes fundamentales: diagnóstico, pruebas de concepto (PoC) y diseño incremental. Esta estructura no surge de manera aislada, sino que se apoya directamente en los contenidos desarrollados en los capítulos previos. El Capítulo 1 aportó el marco conceptual relativo a sistemas *legacy*, sistemas ERP y arquitecturas monolíticas y de microservicios, estableciendo los problemas inherentes a los monolíticos y las oportunidades que brindan los microservicios. El Capítulo 2 incorporó los habilitadores tecnológicos —contenedores, nube, CI/CD y patrones de diseño— que constituyen los medios prácticos para materializar la transición. El Capítulo 3 introdujo principios de diseño esenciales, como la cohesión, el bajo acoplamiento y los mecanismos de comunicación y transacciones distribuidas, los cuales se transforman en criterios rectores para la construcción de los microservicios. Finalmente, el Capítulo 4 presentó tres enfoques para formular microservicios a partir de sistemas monolíticos. Asimismo, se profundizó en el rol de las

PoC y en la conformación de equipos interdisciplinarios, ambos elementos incorporados de manera explícita en la metodología propuesta.

De esta manera, la metodología aquí planteada se erige como la síntesis aplicada de los cuatro primeros capítulos, articulando teoría y práctica en un esquema gradual, evolutivo y adaptable a distintos contextos organizacionales. La figura 14, mediante un esquema gráfico sintetiza la metodología.

Figura 14

Metodología propuesta para el rediseño de un sistema ERP legacy monolítico a microservicios



5.2 Etapa 1 - Elaboración del diagnóstico

La primera etapa consiste en la elaboración de un diagnóstico integral del sistema ERP monolítico. Su objetivo es comprender en profundidad las características, limitaciones y dependencias críticas del sistema *legacy*, de manera de contar con información confiable para planificar la transformación.

En concordancia con el Capítulo 1, el diagnóstico parte de la identificación de las debilidades propias de los sistemas monolíticos —fuerte acoplamiento, dificultad para escalar y limitaciones para adoptar nuevas tecnologías— y del análisis de los módulos funcionales característicos de los ERP. Este reconocimiento permite establecer un mapa detallado de los procesos de negocio, dependencias técnicas y puntos críticos que deben priorizarse en la migración.

5.2.1 Formación un equipo de trabajo interdisciplinario

El diagnóstico debe ser llevado adelante por un equipo interdisciplinario que integre perfiles técnicos (arquitectos de *software*, desarrolladores, administradores de bases de datos) y perfiles funcionales (usuarios de referencia, analistas de procesos). Esta perspectiva plural resulta esencial para garantizar que el análisis no se limite a aspectos tecnológicos, sino que contemple las necesidades estratégicas de la organización.

Tal como se enfatizó en el Capítulo 4, un equipo interdisciplinario constituye un factor clave en esta etapa, ya que asegura que las perspectivas técnicas y funcionales sean consideradas en el análisis inicial.

La selección del personal para la conformación del equipo, debe ser consensuada con la dirección de la empresa, ya que la convocatoria a formar parte del equipo de trabajo, afectará al trabajo realizado en sus áreas específicas

5.2.2 Entregables del diagnóstico

El diagnóstico debe generar un conjunto de entregables que constituyan la base documental y analítica sobre la cual se planifique la transición hacia microservicios. Estos entregables aportan claridad técnica, favorecen la comunicación con las partes interesadas (*stakeholders*) y permiten fundamentar las decisiones posteriores.

A continuación se describe cada uno de ellos:

1. Inventario funcional del sistema

Se elabora un relevamiento exhaustivo de los módulos funcionales que componen el sistema ERP (ventas, compras, contabilidad, gestión de inventarios, entre otros), identificando:

- Procesos de negocio que soportan.
- Usuarios o áreas que los utilizan.
- Nivel de criticidad en la operatoria diaria.
- Interdependencias entre los módulos.
- Frecuencia de modificaciones

Este inventario constituye un mapa funcional que permite visualizar cómo se organiza actualmente el sistema y qué impacto tendría la migración de cada componente.

2. Inventario técnico

Incluye la identificación de las tecnologías que conforman el sistema monolítico, tales como:

- Lenguajes de programación y *frameworks* utilizados.
- Motores de bases de datos, esquemas y volumen de información.
- Dependencias externas (bibliotecas, APIs de terceros, sistemas satélites).

- Infraestructura física o virtual sobre la cual se ejecuta el ERP.

Este inventario es esencial para determinar el grado de obsolescencia tecnológica y evaluar la compatibilidad con arquitecturas de microservicios.

3. Mapa de dependencias internas y externas

A partir de los inventarios funcional y técnico, se confecciona un mapa de dependencias que muestre:

- Llamadas entre módulos internos.
- Interfaz con sistemas externos (por ejemplo: sistemas bancarios, plataformas de *e-commerce*, herramientas de reportes, entre otros).
- Acoplamientos fuertes que podrían dificultar la separación de los módulos.

El objetivo es identificar puntos de fragilidad y definir qué componentes requieren estrategias más cuidadosas de desacoplamiento.

4. Análisis de puntos críticos y cuellos de botella

Se realiza un estudio específico de aquellos módulos que presentan problemas recurrentes, tales como:

- Bajo rendimiento ante sobrecarga.
- Limitaciones de escalabilidad.
- Alta tasa de errores.
- Dependencia de tecnologías obsoletas.

Este análisis permite priorizar los componentes cuya migración generaría un mayor beneficio en términos de estabilidad y performance.

5. Evaluación de calidad de datos

Se analiza la integridad, consistencia y redundancia de los datos administrados por el ERP. Esta información es crucial, dado que la migración hacia microservicios suele implicar la separación de bases de datos y la redefinición de modelos de información.

6. Priorización de módulos candidatos a migrar

A partir de los puntos anteriores, se genera una lista de módulos ordenados por criterios de factibilidad y conveniencia. Esta priorización no es definitiva, pero sirve como guía para definir el alcance de las primeras PoC y orientar el diseño incremental.

7. Documento de conclusiones y recomendaciones iniciales

Finalmente, se genera un informe ejecutivo que sintetiza los hallazgos, expone los puntos más críticos y establece recomendaciones para las etapas siguientes. Este documento constituye un insumo clave para la dirección de la organización, ya que traduce los aspectos técnicos en implicancias estratégicas.

5.3 Etapa 2 - Prueba de conceptos (PoC)

Con el propósito de disminuir la incertidumbre técnica y organizacional que caracteriza la migración de un ERP monolítico hacia una arquitectura basada en microservicios, se plantea la necesidad de realizar un conjunto estructurado de pruebas de concepto PoC (capítulo 4.3). Estas pruebas constituyen ejercicios experimentales acotados que permiten validar, en un entorno controlado, las dimensiones críticas del nuevo diseño, asegurando que las decisiones metodológicas y tecnológicas se fundamenten en evidencia práctica y no únicamente en supuestos teóricos.

En este contexto, las pruebas de concepto no se limitan únicamente a una función de verificación técnica, sino que también desempeñan un papel fundamental en el aprendizaje organizacional. A través de ellas, el equipo de desarrollo adquiere experiencia práctica en el uso de nuevas tecnologías y prácticas de diseño, al tiempo que se genera confianza en las partes interesadas respecto de la viabilidad del proceso de transformación. Asimismo, el orden en que se ejecutan las PoC resulta decisivo, dado que debe responder a una curva de aprendizaje progresiva: en una primera instancia se valida la base tecnológica y operativa; posteriormente se abordan mecanismos transversales críticos, como la comunicación entre servicios, la gestión de transacciones y la observabilidad; y, finalmente, se realizan PoC específicas vinculadas con el enfoque *Strangler Fig*, destinadas a ensayar en pequeña escala la estrategia de migración incremental que se aplicará en etapas posteriores.

5.3.1 Fase 1 – Validación de la infraestructura tecnológica

Las PoC realizadas en esta fase aseguran que la base técnica sea sólida antes de abordar complejidades mayores.

- Validación del *stack* tecnológico: esta prueba consiste en implementar un microservicio prototipo utilizando el lenguaje de programación, *framework* y motor de base de datos seleccionados para la migración. Su objetivo es confirmar la viabilidad técnica de la pila tecnológica propuesta, identificar posibles incompatibilidades y evaluar la curva de aprendizaje del equipo.
- Prueba de contenedorización y orquestación: se construye un servicio en contenedor (por ejemplo, *Docker*) y se despliega en un entorno de orquestación (por ejemplo *Kubernetes*). La finalidad es verificar la facilidad de empaquetado, despliegue, escalabilidad automática y recuperación ante fallos, lo que constituye la base operativa de una arquitectura de microservicios.

5.3.2 Fase 2 – Validación de mecanismos transversales

Las PoC realizadas en esta fase garantizan que los servicios puedan comunicarse, ser escalables y observables.

- Evaluación de la comunicación entre servicios: la PoC aborda mecanismos de comunicación sincrónica (REST[33], gRPC[34]) y asincrónica (mensajería con *RabbitMQ*, *Kafka*, u otros). Su propósito es analizar la latencia, confiabilidad y acoplamiento de los servicios bajo distintas cargas de trabajo, identificando la opción más adecuada para el contexto organizacional.
- Gestión de transacciones distribuidas: dado que los ERP requieren operaciones que afectan múltiples módulos (por ejemplo, compras, inventario y contabilidad), esta prueba implementa patrones como Sagas para garantizar la consistencia eventual de los datos. El resultado esperado es determinar la factibilidad de adoptar un modelo distribuido frente a la rigidez de las transacciones monolíticas ACID.
- Prueba de gestión de datos y autonomía de bases de datos: se experimenta con la separación de esquemas de base de datos entre dos servicios piloto, evaluando los mecanismos de sincronización, consistencia e integridad. Esta PoC permite identificar los riesgos de redundancia y conflictos de datos que surgen en una arquitectura distribuida.
- Implementación de un API *Gateway*: se despliega un API *gateway* que centralice el acceso a múltiples microservicios. El objetivo es validar aspectos como la seguridad (autenticación/autorización), balanceo de carga, limitación de tráfico y enrutamiento de solicitudes, funciones esenciales para el consumo controlado de servicios desde aplicaciones cliente.
- Prueba de monitoreo, trazabilidad y *logging* distribuido: se implementan herramientas de observabilidad (por ejemplo. *Prometheus*, *Grafana*, ELK *stack*, *Jaeger*) sobre un conjunto de servicios prototipo. El propósito es garantizar que la arquitectura resultante cuente con mecanismos adecuados de supervisión y diagnóstico, indispensables para la operación continua.
- Evaluación de escalabilidad y rendimiento: esta prueba mide la capacidad de un servicio prototipo de soportar incrementos de carga mediante escalamiento horizontal. Se busca determinar los límites de desempeño y el costo asociado a mantener niveles adecuados de servicio bajo diferentes escenarios de uso.

[33] REST: (Representational State Transfer) es un estilo de arquitectura para el diseño de sistemas distribuidos, particularmente servicios web, que se basa en un conjunto de principios y restricciones orientados a lograr simplicidad, escalabilidad y desacoplamiento entre clientes y servidores.

[34] gRPC es un framework de comunicación de código abierto, desarrollado por Google, que permite la creación de servicios distribuidos mediante llamadas a procedimientos remotos (RPC, Remote Procedure Call).

- Prueba de tolerancia a fallos y recuperación: se simulan caídas de servicios y de nodos de infraestructura, con el objetivo de evaluar la resiliencia del sistema, la capacidad de recuperación automática y el impacto en la experiencia de usuario.

5.3.3 Fase 3 – Validación del enfoque Strangler Fig

Las PoC realizadas en esta fase permiten ensayar la estrategia de migración progresiva antes de aplicarla en el sistema real.

- PoC de encapsulamiento del monolítico: se implementa un mecanismo de enrutamiento que permita encapsular parte de la lógica del sistema monolítico detrás de un servicio intermediario, como un API *Gateway*. El objetivo es validar la viabilidad de interceptar las solicitudes dirigidas al monolítico y redirigirlas hacia un microservicio recién creado, sin interrumpir la operación existente.
- PoC de extracción de un módulo funcional: se selecciona un módulo de baja criticidad y se implementa como microservicio independiente, utilizando la técnica *Strangler Fig*. Esta prueba permite evaluar los pasos de extracción, la refactorización del monolítico y la interacción del nuevo servicio con el resto de los componentes.
- PoC de coexistencia entre monolítico y microservicio (tráfico selectivo): esta prueba de concepto se orienta a evaluar la capacidad del sistema para operar de manera simultánea con componentes del monolítico y microservicios recientemente extraídos. Para ello, se implementa un mecanismo de enrutamiento selectivo —generalmente a través de un API *Gateway* o *proxy inverso*— que permite dirigir parte del tráfico hacia el módulo original y otra parte hacia el microservicio correspondiente. El objetivo de esta PoC es confirmar que ambos entornos puedan convivir sin afectar la consistencia de los datos ni la experiencia de los usuarios. Asimismo, posibilita identificar riesgos asociados a la duplicación de lógica y establecer estrategias de sincronización durante el período de transición.
- PoC de reemplazo progresivo de funcionalidades (con desactivación en el monolítico): en esta prueba se busca validar la factibilidad de sustituir gradualmente funcionalidades específicas del monolítico por microservicios equivalentes. El procedimiento consiste en desarrollar un microservicio que replique la lógica de un módulo previamente seleccionado, redirigir el tráfico hacia el nuevo componente y, finalmente, proceder a la desactivación controlada de la funcionalidad en el sistema monolítico. La finalidad es verificar que el reemplazo no genere interrupciones en los procesos de negocio, que se preserve la integridad de los datos y que los usuarios no perciban diferencias en el uso del sistema. De esta manera, se ensaya de forma acotada la estrategia de migración incremental, asegurando que cada sustitución se realice con el menor impacto posible.

5.3.4 Fase 4 – Validación organizacional

La última fase de las pruebas de conceptos se orienta a la dimensión organizacional de la migración. Aun cuando los aspectos técnicos resulten exitosos, un proceso de transformación hacia microservicios puede fracasar si los usuarios y las partes interesadas no comprenden ni validan los beneficios de la nueva arquitectura. Por esta razón, se plantea la necesidad de realizar PoC enfocadas en la experiencia de usuario, la alineación con los procesos de negocio y la aceptación de la dirección de la organización.

- Validación con stakeholders: esta PoC consiste en seleccionar una funcionalidad representativa del ERP —preferentemente de mediana criticidad y de uso frecuente— e implementarla como microservicio en un entorno de prueba accesible a usuarios finales y responsables de negocio. El objetivo es que los actores clave puedan interactuar directamente con el nuevo componente y contrastarlo con la funcionalidad equivalente en el monolítico.

5.4 Etapa 3 – Diseño incremental

La tercera etapa de la metodología constituye el núcleo del proceso de transformación, dado que se orienta a la implementación gradual de la arquitectura de microservicios. Esta etapa se fundamenta en el enfoque *Strangler Fig*, mencionado en el capítulo 4. La elección de este enfoque responde a la necesidad de minimizar riesgos, garantizar la continuidad operativa y permitir ajustes iterativos, en contraposición con estrategias como el *Big Bang*, que implican altos niveles de incertidumbre y altas posibilidades de fracaso.

5.4.1 Criterios de extracción de módulos

La selección de los módulos a migrar desde el monolítico a microservicios debe basarse en un conjunto de criterios que integren tanto dimensiones técnicas como de negocio. Los principales enfoques son los siguientes:

- Basado en dominios del negocio: siguiendo los lineamientos del diseño dirigido por el dominio DDD (capítulo 4.1), se extraen primero los módulos que representan procesos estratégicos del ERP, asegurando que la arquitectura refleje las prioridades organizacionales.
- Basado en escalabilidad y rendimiento: se priorizan aquellos módulos que generan cuellos de botella o que requieren gran capacidad de procesamiento, ya que su migración puede generar beneficios inmediatos en la performance global.
- Basado en la frecuencia del cambio: se seleccionan módulos sujetos a modificaciones recurrentes por normativas o cambios de mercado, de modo que la transición a microservicios incremente la agilidad de la actualización.

- Basado en la autonomía de datos: se extraen en primera instancia los módulos que gestionan datos relativamente independientes, reduciendo la complejidad de la separación de bases de datos y la necesidad de transacciones distribuidas.
- Basado en fronteras de comunicación: se priorizan aquellos módulos que ya interactúan con el resto del sistema mediante interfaces bien definidas, lo cual facilita su encapsulamiento como microservicios autónomos.

La migración efectiva requiere combinar varios enfoques para equilibrar la factibilidad técnica con el impacto organizacional.

5.4.2 Proceso iterativo de diseño incremental

El proceso de diseño incremental se concibe como un conjunto de ciclos sucesivos, en los que se planifica, desarrolla, despliega y valida la extracción de funcionalidades del monolítico hacia microservicios. Cada ciclo aporta valor inmediato y, al mismo tiempo, sienta las bases para la migración de componentes adicionales. De esta manera, se asegura que la organización avance de forma controlada, mitigando riesgos y capitalizando el aprendizaje obtenido en cada iteración. Los pasos fundamentales de este proceso se describen a continuación:

1. Selección del módulo a migrar: en cada iteración se elige un módulo del monolítico considerando los criterios definidos previamente (dominios de negocio, rendimiento, frecuencia de cambio, autonomía de datos o fronteras de comunicación). La elección estratégica de los módulos iniciales resulta determinante, ya que un error en esta etapa podría generar sobrecostos o retrasos en el cronograma de migración.
2. Diseño y especificación del microservicio: una vez seleccionado el módulo, se define el nuevo microservicio. Esto implica delimitar con precisión sus responsabilidades, elaborar el modelo de datos correspondiente y establecer contratos de API claros para la comunicación con otros servicios y el monolítico. En este punto, resulta fundamental aplicar principios de cohesión y bajo acoplamiento, desarrollados en el capítulo 3, a fin de asegurar que el nuevo servicio sea verdaderamente autónomo.
3. Desarrollo e implementación del microservicio: el microservicio se construye utilizando la pila tecnológica definida en las etapas previas y validada mediante las PoC. Se implementan patrones de diseño que refuercen la resiliencia, como *Circuit Breaker* o *Retry*, y se integran prácticas de desarrollo ágil, con énfasis en la automatización de pruebas.
4. Contenerización y despliegue en entorno de orquestación: el microservicio se empaqueta en contenedores (*Docker*) y se despliega en un entorno de orquestación (*Kubernetes* u otro equivalente). Este paso asegura la portabilidad, el escalado dinámico y la capacidad de recuperación ante fallos.

5. Integración con el sistema existente mediante mecanismos de enrutamiento: se configura un API *Gateway* que permita interceptar las solicitudes dirigidas al monolítico y derivarlas hacia el nuevo microservicio. Esta técnica permite introducir gradualmente el microservicio en la operación sin necesidad de interrumpir el sistema *legacy*.
6. Validación funcional y de rendimiento: el nuevo microservicio se somete a pruebas para garantizar que cumple con los mismos requerimientos funcionales que el módulo original. Asimismo, se realizan pruebas de rendimiento y escalabilidad que permitan medir mejoras respecto al componente previo. Esta validación constituye un hito clave en cada iteración, dado que define si es posible continuar hacia la desactivación del módulo en el monolítico.
7. Monitoreo post-migración: una vez puesto en funcionamiento, se monitorea el microservicio mediante métricas de disponibilidad, latencia, consumo de recursos y tasa de errores. El monitoreo es indispensable para detectar incidentes tempranos y asegurar que la transición no degrade la calidad del servicio.
8. Desactivación controlada de la funcionalidad en el monolítico: tras comprobar la estabilidad y el correcto funcionamiento del nuevo microservicio, se procede a deshabilitar gradualmente la lógica equivalente en el monolítico.
9. Retroalimentación y ajuste del proceso: cada ciclo culmina con un análisis de resultados y lecciones aprendidas. La retroalimentación obtenida se incorpora en las iteraciones posteriores, lo que permite perfeccionar la estrategia de extracción, ajustar los criterios de priorización y mejorar la eficiencia del proceso global de migración.

El proceso iterativo de diseño incremental constituye, en definitiva, la pieza central de la transición hacia microservicios. Su naturaleza cíclica permite que cada avance se convierta en un logro tangible, aportando valor inmediato a la organización y reduciendo, al mismo tiempo, los riesgos inherentes a la migración de sistemas complejos como los ERP. Al incorporar mecanismos de selección estratégica, validación continua, monitoreo y retroalimentación, la metodología asegura que la transformación no sea concebida como un evento único, sino como una secuencia de mejoras progresivas que fortalecen tanto la arquitectura tecnológica como las capacidades organizacionales.

De este modo, el enfoque incremental basado en el patrón *Strangler Fig,* (capítulo 4.2.2), no solo ofrece un camino seguro y controlado para la modernización del sistema *legacy*, sino que también habilita a la organización a mejorar, adaptarse y evolucionar en cada iteración, garantizando que el resultado final responda de manera efectiva a los objetivos estratégicos y a las exigencias dinámicas del entorno competitivo.

Conclusión

La presente investigación permitió analizar en profundidad los desafíos y oportunidades asociados a la transición de un sistema monolítico hacia una arquitectura basada en microservicios, con especial énfasis en los sistemas ERP. Este abordaje teórico y metodológico posibilitó alcanzar el objetivo general de la tesina, consistente en proponer una metodología de trabajo para abordar el rediseño de un sistema ERP *legacy* monolítico hacia una arquitectura de microservicios.

El desarrollo del marco conceptual abordado en los primeros capítulos permitió, en consonancia con los objetivos secundarios, investigar los sistemas *legacy*, los sistemas ERP, las arquitecturas monolíticas y las arquitecturas de microservicios, así como analizar las ventajas, limitaciones y características de cada enfoque. Esta revisión teórica permitió comprender que, si bien los sistemas monolíticos han demostrado ser sólidos y estables, presentan limitaciones significativas en cuanto a escalabilidad, flexibilidad y capacidad de adaptación a entornos dinámicos. En contraposición, las arquitecturas de microservicios emergen como una alternativa viable para superar estas restricciones, al promover la modularidad, la autonomía funcional y la capacidad de adoptar tecnologías heterogéneas según los requerimientos de cada servicio.

Asimismo, el trabajo permitió establecer criterios para formular microservicios a partir del monolítico, apoyándose en enfoques como el diseño dirigido por el dominio, la identificación de contextos delimitados, el análisis de la autonomía de datos y la priorización según necesidades de negocio o rendimiento. Del mismo modo, se cumplió con el objetivo de analizar los patrones de diseño de microservicios que contribuyen al rediseño del sistema, tales como *Strangler Fig*, *Branch by Abstraction*, *Parallel Run, API Gateway*, *Circuit Breaker* y *Retry*, los cuales demostraron ser fundamentales para garantizar una transición segura, gradual y resiliente.

En este marco, la metodología presentada en el capítulo 5 integró de manera explícita todos estos aportes, dando cumplimiento al objetivo de diseñar las etapas de un proceso metodológico orientado al rediseño del ERP *legacy*. La metodología propuesta organiza el proceso en tres etapas complementarias: el diagnóstico inicial, orientado a comprender las características, dependencias y limitaciones del sistema ERP *legacy*; las pruebas de concepto, destinadas a reducir incertidumbre tecnológica y organizacional; y el diseño incremental basado en el patrón *Strangler Fig*, que posibilita la migración progresiva y controlada de funcionalidades. Esta estrategia se presenta como una alternativa superadora frente a enfoques de reemplazo abrupto, al permitir reducir riesgos, asegurar continuidad operativa y capitalizar aprendizajes en cada iteración.

Finalmente, la investigación permitió concluir que la transición hacia una arquitectura de microservicios no depende únicamente de la aplicación rigurosa de

técnicas y patrones de diseño, sino también de la articulación entre aspectos tecnológicos, organizacionales y estratégicos. El cambio arquitectónico debe entenderse como un proceso integral que incide en la cultura organizacional, en las prácticas de desarrollo y en la capacidad de innovación futura. En este sentido, la metodología propuesta constituye un aporte concreto y fundamentado que puede ser utilizado como guía para organizaciones que enfrentan el desafío de modernizar sus sistemas ERP monolíticos.

Futuras investigaciones

A partir de los resultados alcanzados y de la metodología propuesta para la migración de sistemas ERP *legacy* monolíticos hacia arquitecturas basadas en microservicios, se identifican diversas líneas de investigación futura, que permitirían profundizar y ampliar los aportes de este trabajo:

1. Validación empírica de la metodología: la aplicación de la metodología propuesta en casos reales de migración de ERP en distintas organizaciones permitiría contrastar su efectividad en contextos heterogéneos. Un estudio comparativo de casos podría aportar evidencia sobre los beneficios y dificultades experimentadas en cada etapa del proceso.
2. Automatización de la identificación de módulos candidatos: investigar técnicas de *machine learning* o minería de *softwar*e para analizar la estructura del monolítico y sugerir automáticamente módulos susceptibles de extracción. Este enfoque podría optimizar el diagnóstico y reducir la subjetividad en la priorización de componentes.
3. Impacto organizacional de la migración: profundizar en estudios sobre la gestión del cambio, la capacitación de equipos y la aceptación de usuarios durante la transición. Un análisis sociotécnico permitiría comprender cómo factores humanos y culturales influyen en el éxito de la modernización arquitectónica.
4. Modelos de costos y beneficios en la migración: diseñar modelos que cuantifiquen los costos asociados al rediseño (infraestructura, capacitación, tiempo de inactividad) y los comparen con los beneficios esperados (escalabilidad, flexibilidad, innovación). Esta línea de investigación facilitaría la toma de decisiones estratégicas en organizaciones con recursos limitados.

Bibliografia

1. Ulrich, W. M. (1994). *Legacy systems: transformation strategies*. Prentice Hall.
2. Bennett, K. (1995). Legacy systems: coping with success. *IEEE Software*, 12(1), 19-23.
3. Esteves, J., & Pastor, J. (1999). An ERP lifecycle-based research agenda.En *1ºInternational Workshop on Enterprise Management Resource and Planning Systems (EMRPS)* 359-371.
4. Fowler, M. (2014). *Microservices*. Recuperado de https://martinfowler.com/articles/microservices.html
5. Oracle. (2024). *ERP.* Recuperado de https://www.oracle.com/ar/erp
6. Richardson, C., & Smith, F. (2016). *Microservices from design to deployment*. NGINX. Recuperado de https://www.f5.com/content/dam/f5/corp/global/pdf/ebooks/Microservices_Designing_Deploying.pdf
7. Daya, S., Van Duy, N., Eati, K., Ferreira, C. M., Lozic, D., Gucer, V., Gupta, M., Joshi, S., Lampkin, V., Martins, M., Narain, S., & Vennam, R. (2015). *Microservices from theory to practice: Creating applications in IBM Bluemix using the microservices approach.* Recuperado de https://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf
8. Newman, S. (2021). *Building microservices: Designing fine-grained systems*. O'Reilly Media
9. Newman, S. (2020). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media
10. Bonér, J. (2016). *Reactive microservices architecture: Design principles for distributed systems*. O'Reilly Media
11. Oracle. (2024). *What is cloud computing?.* Recuperado de https://www.oracle.com/ar/cloud/what-is-cloud-computing/#types-of-cloud-computing
12. DeMarco, T. (1979). *Structured Analysis and System Specification.*Prentice Hall.
13. Myers, G. J. (1978). *Composite/structured design*. Van Nostrand Rinhold.
14. Page Jones, M. (1980). *The practical guide to structured systems design*. Prentice Hall.
15. Constantine, L., & Yourdon, E. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Yourdon Press
16. Schach, S. R. (1991). *Classical and object-oriented software engineering*. MacGraw-Hill.
17. Google Cloud. (s.f.). *What are containers?*. Recuperado de https://cloud.google.com/learn/what-are-containers

18. Marteens, I. (1998). *La cara oculta de Delphi 4*. Anaya Multimedia.
19. Date, C. J. (2001). *Introducción a los Sistema de bases de datos* (7ª ed.). Pearson Educación.
20. Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison Wesley
21. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
22. Conway, M. (s.f.). *Consway´s Law*. Recuperado de https://www.melconway.com/Home/Conways_Law.html
23. Fowler, M. (2022). *Consway´s Law*. Recuperado de https://martinfowler.com/bliki/ConwaysLaw.html
24. Nygard, M. T. (2017). *Release It!: Design and deploy production-ready software* (2ª ed.). Pragmatic Programmers.
25. Hernández Sampieri, R. (2014). *Metodología de la investigación* (6ª ed.) MacGraw-Hill.
26. Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 249–259. Association for Computing Machinery. Recuperado de https://doi.org/10.1145/38714.38742.
27. Spartan Cybersecurity. (2025). *Introducción a los contenedores y la orquestación*. Recuperado de https://books.spartan-cybersec.com/cpna/introduccion-a-ecs-eks-ecr/introduccion-a-los-contenedores-y-la-orquestacion
28. Amazon Web Services. (2025). *¿Qué es la orquestación de contenedores?* Recuperado de https://aws.amazon.com/es/what-is/container-orchestration/
29. Whitestack. (2025). *¿Qué es un sistema legacy?* Whitestack. Recuperado de https://whitestack.com/es/blog/sistema-legacy
30. GeneXus. (2025). *Modernización de sistemas legados*. GeneXus. Recuperado de https://www.genexus.com/es/productos/modernizacion-de-sistemas-legados
31. Red Hat. (s.f.). *¿Qué es CI/CD?* Red Hat. https://www.redhat.com/es/topics/devops/what-is-ci-cd
32. Invgate. (2025). *Stack tecnológico: bases sólidas para el desarrollo de software*. Recuperado de https://blog.invgate.com/es/stack-tecnologico
33. Santana, D. (2023). *Cloud computing demystified for aspiring professionals*. Packt Publishing.
34. Microsoft. (2022). *NET microservices: Architecture for containerized .NET applications* (6th ed.). Microsoft Corporation. Recuperado de https://docs.microsoft.com/dotnet/architecture/microservices/

35. Microsoft. (2025). *Patrón Retry*. Azure Architecture Center. Recuperado de https://learn.microsoft.com/es-es/azure/architecture/patterns/retry
36. GeeksforGeeks. (2025). What is Proof of Concept (POC) in Software Development?. Recuperado de https://www.geeksforgeeks.org/what-is-proof-of-concept-poc-in-software-development/
37. Atlassian. (2025). *Your guide to proof of concept (POC) in product development*. Recuperado de https://www.atlassian.com/work-management/project-management/proof-of-concept
38. Raeburn, A. (2025). *Equipos interdisciplinarios: 9 consejos y beneficios*. Asana. Recuperado de https://asana.com/es/resources/cross-functional-team
39. Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.
40. Blank, S. (2013). *The Startup Owner's Manual: The Step-by-Step Guide for Building a Great Company*. K&S Ranch.

Aval del director de tesis – FCyT 2025

Estudiante/s: Sergio Ricardo Montero

Título del trabajo de investigación: Proponer una metodología para rediseñar un sistema ERP legacy monolítico hacia una arquitectura de microservicios.

Programa de estudios: **Licenciatura en Sistemas de Información**

Fecha de entrega: 09/12/2025

Por medio de la presente, me dirijo a ustedes en calidad de Director de Tesis a fin de dar consentimiento respecto al cumplimiento de los requisitos de escritura y calidad académica del trabajo final del cual presido, en conformidad con lo establecido en la "Resolución CD FCyT Nro. 503/23".

Ademas, quiero dejar en claro que tanto los objetivos propuestos en el pre-proyecto aprobado en el año lectivo 2022 fueron respetados y llevados a cabo en todo el trabajo de investigación desarrollado.-

Ing. Pablo Emanuel Goette
goette.pablo@uader.edu.ar