UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

MAESTRÍA EN

SOFTWARE CON MENCIÓN EN DISEÑO DE ARQUITECTURA DE SISTEMAS

RPC-SO-34-NO.778-2021

OPCIÓN DE TITULACIÓN:

ARTÍCULOS PROFESIONALES DE ALTO NIVEL

TEMA:

ESTRATEGIAS PARA TRANSFORMAR APLICACIONES MONOLÍTICAS A APLICACIONES DISTRIBUIDAS ALTAMENTE DESACOPLADAS

AUTOR:

HÉCTOR DAVID ANDRADE UNUSUNGO

DIRECTOR:

CHRISTIAN MAURICIO MERCHÁN MILLÁN

QUITO – ECUADOR

2025

Autor(es):

Héctor David Andrade Unusungo
Ingeniero en Electrónica y Redes de Información por la Escuela Politécnica Nacional
Candidato a Magíster en Software con Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Quito.
handradeu@est.ups.edu.ec

Dirigido por:

Christian Mauricio Merchán Millán
Ingeniero en Computación por la Escuela Superior Politécnica del Litoral
Máster en Sistemas de Información Gerencial por la Escuela Superior Politécnica del Litoral
cmerchan@ups.edu.ec

Héctor David Andrade Unusungo
ESTRATEGIAS PARA TRANSFORMAR APLICACIONES MONOLÍTICAS A APLICACIONES DISTRIBUIDAS ALTAMENTE DESACOPLADAS

Estrategias para transformar aplicaciones monolíticas a aplicaciones distribuidas altamente desacopladas

Héctor David Andrade Unusungo

Universidad Politécnica Salesiana - Ecuador

Resumen

En el ámbito del desarrollo de software, es común encontrar sistemas basados en arquitecturas monolíticas debido a su simplicidad inicial y facilidad de implementación. Sin embargo, a medida que el software evoluciona y crece en complejidad, los problemas asociados con esta arquitectura suelen superar sus beneficios, lo que demanda decisiones estratégicas para guiar su evolución. La transformación de una arquitectura monolítica hacia estilos más desacoplados presenta tanto desafíos técnicos como de negocio, que deben abordarse de manera integral. Este trabajo aborda dicho proceso de evolución mediante la implementación y análisis de varias pruebas de concepto. En el primer caso, se lleva a cabo la transformación de una aplicación monolítica hacia una arquitectura orientada a eventos (*Event-Driven Architecture*, EDA), lo que permite desacoplar los servicios y mejorar la escalabilidad. En el segundo caso, se implementa un enfoque basado en *Command Query Responsibility Segregation* (CQRS) en conjunto con *Event Sourcing*, para optimizar la separación de responsabilidades y mejorar el rendimiento en entornos de alta concurrencia. Finalmente, en el tercer caso, la aplicación monolítica es descompuesta siguiendo los principios de la arquitectura de microservicios, lo que facilita su escalabilidad, mantenimiento y resiliencia.

Abstract

In the field of software development, monolithic architectures are commonly found due to their initial simplicity and ease of implementation. However, as software evolves and grows in complexity, the challenges associated with this architecture often outweigh its advantages, necessitating strategic decisions to guide its evolution. Transforming a monolithic architecture into more decoupled styles presents both technical and business challenges that must be addressed comprehensively. This work tackles the evolution process through the implementation and analysis of several proof-of-concept cases. In the first case, a monolithic application is transformed into an Event-Driven Architecture (EDA), enabling service decoupling and improved scalability. In the second case, an approach based on Command Query Responsibility Segregation (CQRS) is implemented alongside Event Sourcing, optimizing the separation of concerns and enhancing performance in high-concurrency environments. Finally, in the third case, the monolithic application is decomposed following microservices architecture principles, facilitating its scalability, maintainability, and resilience.

Palabras clave— Aplicaciones monolíticas, Aplicaciones distribuidas altamente desacopladas, Event Driven Architecture, CQRS, Event Sourcing, Axon, Microservicios, RabbitMQ, Consul, Apache Kafka .

1 Introducción

Durante mucho tiempo, las arquitecturas monolíticas fueron el estándar de facto para implementar aplicaciones y sistemas en general debido a su simplicidad y eficiencia inicial. Sin embargo, a medida que las demandas de los usuarios y las condiciones del mercado evolucionan, este enfoque arquitectónico ha enfrentado desafíos crecientes en términos de escalabilidad, mantenibilidad y flexibilidad Dragoni et al. (2017).

Considerando lo anterior, en la última década se ha evidenciado una transformación significativa en la industria del software en cuanto a cómo se diseñan, desarrollan y despliegan las aplicaciones Lisa J. Kirby et al. (2021). La adopción de metodologías ágiles, el surgimiento de la infraestructura en la nube, y la popularización de tecnologías tales como la contenerización de aplicaciones y la orquestación de sus componentes son ejemplos de cómo está cambiando la forma en que los desarrolladores abordan el ciclo de vida del software Sharma et al. (2020); Younas et al. (2020). No obstante, transformar aplicaciones monolíticas en aplicaciones de componentes altamente desacoplados para mejorar su desempeño y escalabilidad no es un proceso trivial. Este estudio busca abordar el desafío de cómo realizar esta transformación, explorando estrategias y marcos de trabajo para ejecutarla de manera efectiva y eficiente con un enfoque práctico basado en pruebas de concepto reales.

La investigación en torno a la transformación de arquitecturas monolíticas hacia sistemas distribuidos y altamente desacoplados ha sido extensa y diversa. Un enfoque ampliamente discutido es la Arquitectura Orientada a Servicios (SOA), la cual, según Clark and Barn (2012) se basa en la exposición de funcionalidades empresariales a través de interfaces bien definidas, lo que permite que los componentes interactúen mediante mensajes, ya sea de manera síncrona o asíncrona. Esta idea se refuerza con lo indicado por Oliveira et al. (2018) quien destaca las ventajas de SOA para promover la reusabilidad, interoperabilidad y una mejor gestión de los servicios dentro de una organización. Todas estas características permiten crear una estructura flexible que facilita la evolución del sistema y la integración de nuevos servicios sin impactar significativamente en las operaciones existentes.

Sin embargo, SOA no es la única alternativa para abordar el problema del acoplamiento en sistemas monolíticos. Arquitecturas como los Sistemas Autónomos (*Self-Contained Systems*, SCS) y las Arquitecturas Dirigidas por Eventos (*Event-Driven Architectures*, EDA) también pueden aportar soluciones Cerny et al. (2017). Según Trabelsi et al. (2023), EDA permite que los componentes interactúen a través de eventos, reduciendo así la dependencia directa entre ellos y facilitando una mayor escalabilidad y flexibilidad. Por su parte Cabane and Farias (2024) muestra que estos enfoques son especialmente útiles en escenarios donde la capacidad de responder a eventos de manera asíncrona y la optimización de recursos bajo demanda son críticos.

En el contexto del diseño de sistemas distribuidos, CQRS (*Command Query Responsibility Segregation*) y *Event Sourcing* son dos patrones arquitectónicos que, al igual que EDA, promueven el desacoplamiento y la escalabilidad de aplicaciones complejas. De acuerdo con Long (2017) CQRS separa las responsabilidades de lectura y escritura en dos modelos distintos, lo que permite optimizar cada uno para sus respectivas tareas. Esto mejora el rendimiento y facilita la evolución del sistema, ya que los comandos que modifican el estado y las consultas que lo leen pueden escalarse de forma independiente.

Por otro lado, Overeem et al. (2017) describe a CQRS como un paso intermedio hacia *Event Sourcing*, el cual almacena el estado de un sistema como una secuencia de eventos que han ocurrido en lugar de un estado actual, lo que facilita la auditoría, la capacidad de revertir cambios y la recuperación ante fallos. La combinación de ambos patrones proporciona condiciones favorables para construir sistemas desacoplados y resilientes.

En este punto es importante mencionar también que, en los últimos años, el interés por los microservicios ha ido en aumento; Aggarwal and Singh (2024) define a los microservicios como una arquitectura que fragmenta aplicaciones monolíticas en servicios pequeños, independientes y autónomos. Estos microservicios están diseñados para gestionar una única responsabilidad dentro del sistema, lo que permite una mayor modularidad y flexibilidad en el desarrollo. Valentina Lenarduzzi et al. (2020) destaca además que cada microservicio puede ser desarrollado, desplegado y escalado de forma independiente, lo que reduce los riesgos de fallos sistémicos al limitar el impacto a un solo servicio en lugar de afectar a toda la aplicación. Este enfoque promueve un ciclo de desarrollo más ágil y facilita la adopción de nuevas tecnologías dentro de servicios específicos, sin necesidad de modificar el sistema en su totalidad.

Gao et al. (2024) señala en su trabajo cómo esta arquitectura permite una mayor resiliencia y escalabilidad en entornos altamente demandantes. Al estar desacoplados, los microservicios no dependen de una infraestructura centralizada, lo que significa que, si un componente falla, el resto del sistema puede

seguir funcionando sin interrupciones. Esto mejora la tolerancia a fallos y permite un manejo más eficiente de los recursos, ya que cada servicio puede escalarse según la demanda específica. Además, Lau et al. (2024) subraya que los microservicios promueven la autonomía en los equipos de desarrollo, permitiendo la implementación y mantenimiento de servicios individuales sin generar dependencias cruzadas complejas, lo que resulta en una mayor velocidad y eficiencia en el desarrollo de software.

Por lo expuesto anteriormente, es evidente que existe una amplia variedad de enfoques arquitectónicos y estrategias de transformación, lo que resalta la importancia de seleccionar la opción más adecuada en función de las necesidades específicas de cada proyecto. En consecuencia, este trabajo tiene como objetivo ofrecer una perspectiva más amplia, describiendo y comparando diversas estrategias para transformar aplicaciones monolíticas tradicionales en sistemas de componentes altamente desacoplados.

Para alcanzar este objetivo las pruebas de concepto implementadas ejemplifican situaciones basadas en problemas del mundo real, proporcionando soluciones concretas a través de la aplicación de estrategias específicas y de marcos de software, que son descritas en cada caso, aportando un enfoque práctico y basado en evidencia. También se proporciona recomendaciones útiles que pueden guiar la toma de decisiones arquitectónicas en futuros proyectos.

De aquí en adelante, este artículo está organizado de la siguiente manera. En la Sección 2, se describe en términos generales el proceso realizado para seleccionar, analizar y transformar las aplicaciones monolíticas hacia arquitecturas desacopladas. En la Sección 3,4 y 5 se describen los detalles de la cada prueba de concepto, donde se incluye el análisis de los casos, diseño de las soluciones, desarrollo, evaluación de los sistemas y discusión de los resultados. Finalmente, en la Sección 6, se presentan las conclusiones y se ofrecen recomendaciones para futuras investigaciones y prácticas relacionadas con este trabajo.

2 Metodología

Para el presente estudio se han diseñado tres pruebas de concepto, con el objetivo de evaluar y comparar diversas estrategias de transformación de aplicaciones monolíticas a arquitecturas distribuidas altamente desacopladas. La metodología adoptada sigue un enfoque iterativo, utilizando el marco de trabajo Kanban para la gestión de tareas, donde los experimentos fueron ejecutados en entornos controlados para asegurar la replicabilidad de los resultados. Cada una de las estrategias de transformación fue implementada y evaluada en función de su impacto en el rendimiento y la escalabilidad de las aplicaciones.

Cada transformación se dividió en múltiples fases, que incluyeron:

1. Identificación de componentes de la aplicación monolítica.
2. Implementación de estrategias para el desacoplamiento de la aplicación.
3. Integración de componentes desacoplados en función de la estrategia seleccionada.
4. Validación y optimización de la comunicación entre componentes.

Para cada fase, se estableció un conjunto de tareas, que fueron priorizadas según el impacto esperado en la descomposición de la aplicación y su transición hacia una arquitectura distribuida.

Una vez finalizadas las implementaciones, se tabularon los resultados obtenidos para cada estrategia, considerando indicadores de rendimiento como latencia, uso de recursos y *throughput*. Los datos fueron utilizados para comparar el desempeño de las estrategias respecto a su implementación original, determinando los escenarios en los que una arquitectura monolítica es suficiente y aquellos en los que es necesario evolucionarla hacia una arquitectura distribuida. Esta información puede ser utilizada como referencia para la selección de las estrategías descritas al abordar escenarios similares.

3 Caso 1: Comercio Electrónico

3.1 Análisis

3.1.1 Descripción del caso

La primera prueba de concepto se centra en el análisis de un sistema monolítico desarrollado en .NET que forma parte de una aplicación de *e-commerce* y que implementa una arquitectura basada en servicios REST. Este sistema está compuesto por tres controladores principales:

- `CartController`: Gestiona las operaciones relacionadas con el carrito de compras.
- `ProductController`: Gestiona las operaciones de productos, incluyendo su administración y consulta.
- `OrderController`: Maneja las solicitudes de órdenes, integrando servicios internos para la reserva y actualización de inventario, procesamiento de pagos y notificación al cliente.

Además, el sistema integra tres servicios críticos:

- `InventoryService`: Encargado de la reserva y actualización del inventario cuando se genera una orden.
- `PaymentService`: Responsable de procesar los pagos, dependiente de proveedores externos.
- `NotificationService`: Envía notificaciones al cliente cuando una orden ha sido procesada exitosamente.

3.1.2 Problemas detectados

Uno de los problemas más críticos del sistema es el elevado tiempo de espera en la solicitud de órdenes a través del controlador `OrderController`, lo que afecta directamente la experiencia del usuario. Este problema se origina en la naturaleza síncrona de la aplicación, lo cual obliga a que todas las operaciones internas (reserva de inventario, ejecución de pago y notificación) deben completarse antes de devolver una respuesta al cliente.

Por otro lado, cuando se recibe una gran cantidad de solicitudes de órdenes simultáneamente, las demoras en el procesamiento de pagos a través de `PaymentService` pueden generar fluctuaciones en el tiempo de respuesta, debido principalmente a que se trata de servicios externos sobre los cuales el sistema no tiene control, lo cual incrementa potencialmente el riesgo de errores por *timeout* lo que eventualmente terminará afectando la fiabilidad del sistema.

Finalmente, para mantener la consistencia del inventario el sistema implementa un mecanismo de bloqueo, por medio de la clase `System.Threading.Lock`, lo cual garantiza que ningún subproceso pueda acceder al recurso hasta que sea liberado. Si bien esto funciona con un nivel bajo de solicitudes, en condiciones de alta demanda no solo introduce un cuello de botella en `InventoryService`, sino que también incrementa el tiempo total de respuesta, lo cual afecta la experiencia del usuario final y en general la estabilidad del sistema.

3.2 Diseño

Con base en las limitaciones identificadas en el análisis del sistema monolítico y considerando que la comunicación en los flujos descritos pueden modelarse como una secuencia de eventos, se decidió descomponer y migrar parte de la implementación del controlador `OrderController` hacia un modelo asíncrono mediante la introducción de una arquitectura orientada a eventos (*Event Driven Architecture*).

La estrategia principal consistió en reestructurar las operaciones críticas, tales como reserva de inventario, ejecución de pago y notificación al cliente, de forma que no dependieran de un flujo síncrono de control, evitando los cuellos de botella y tiempos de espera extendidos observados en el diseño original. De esta forma se establece una separación de responsabilidades en la cual `OrderController` deja de ser el orquestador directo de las operaciones críticas, distribuyéndolas entre los componentes de la aplicación de la siguiente manera:

- `OrderController` ahora actúa únicamente como el punto de entrada para las solicitudes de órdenes, delegando la coordinación de las operaciones a un *message broker*.
- `InventoryService`, `PaymentService` y `NotificationService` se convirtieron en componentes autónomos e independientes, encargados exclusivamente de manejar las operaciones dentro de su ámbito, y escuchando los eventos relevantes enviados a través del *message broker*.
- *Message broker* es el componente principal en la arquitectura orientada a eventos, siendo el responsable de gestionar la comunicación entre los servicios de manera desacoplada. Se definieron las siguientes colas para el *message broker*, donde cada una tiene una función específica para gestionar el ciclo de vida de una orden:

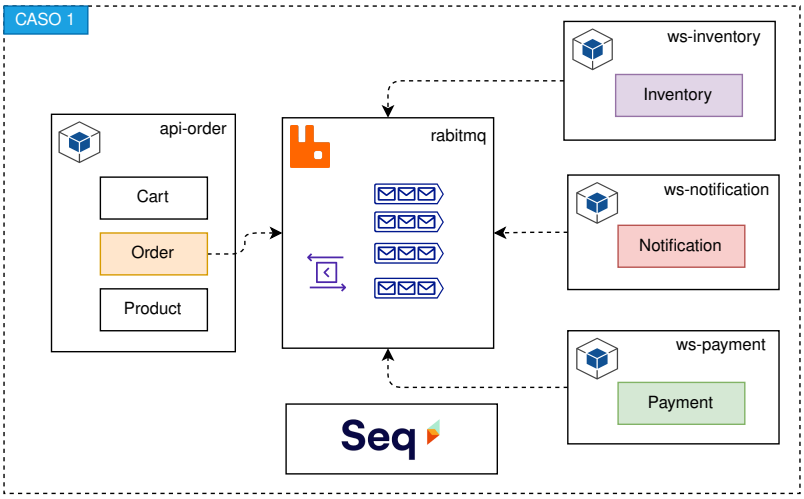

Figura 1: Componentes del sistema monolítico y descomposición con *Event Driven Architecture*

- Cola de solicitudes de órdenes (`order_queue`): Recibe todas las solicitudes generadas por el cliente a través de `OrderController`.
- Cola de reserva de inventario (`inventory_reserved_queue`): Almacena eventos relacionados con la reserva y actualización de inventario.
- Cola de errores de inventario (`inventory_error_queue`): Gestiona eventos relacionados con fallos en el proceso de reserva de inventario.
- Cola de confirmación de pago (`inventory_confirmed_queue`): Almacena eventos de órdenes cuyo pago ha sido confirmado exitosamente.
- Cola de notificación (`notification_queue`): Encargada de notificar al cliente una vez que el proceso ha sido completado.

Esta nueva configuración, representada en la Figura 1, permite gestionar las interacciones de forma asíncrona, lo que elimina la necesidad de que el cliente espere una respuesta inmediata para cada paso del proceso.

Adicionalmente se incorporó un sistema de monitoreo para supervisar el estado de cada componente.

3.3 Desarrollo

En esta etapa se desarrollaron los componentes definidos en el diseño. A continuación, se destacan los aspectos más relevantes de la implementación:

Message broker: Se seleccionó a RabbitMQ como sistema de mensajería debido a su capacidad para gestionar colas de mensajes de manera eficiente y su compatibilidad con .NET. Su integración permitió el manejo de mensajes asíncronos y por medio de la configuración de *durable queues* se garantiza la integridad de los datos en caso de reinicios o fallos en el sistema.

Se configuraron las colas definidas en el diseño utilizando archivos `definitions.json`, evitando que las configuraciones de infraestructura fueran manejadas directamente desde el código. Adicionalmente, para que `inventory_confirmed_queue` y `notification_queue` reciban los eventos correspondientes, se agregó un `payment_exchange` para distribuir estos eventos de forma simultánea. Por otro lado, para garantizar una alta disponibilidad y capacidad de manejo de conexiones bajo alta demanda, se configuraron los parámetros `connection_timeout`, `handshake_timeout` y `tcp_listen_options (backlog)` en `rabbitmq.config`.

Servicios distribuidos: Los componentes `InventoryService`, `PaymentService` y `NotificationService`, se implementaron utilizando *Worker Services* de .NET. Internamente, estos servicios hacen uso de `BackgroundService`, lo que permite la ejecución de tareas prolongadas en segundo plano, permitiendo que cada servicio pueda escuchar continuamente los eventos en sus colas respectivas.

Tabla 1: Características de los sistemas - Caso 1

	MONO1	***EDA***
Lenguaje	C#	C#
Framework	.NET 8	.NET 8
Número de archivos	22	68
Líneas de código efectivas	433	1830
Comentarios	28	88
Líneas en blanco	70	243
Total de líneas	**531**	**2161**

Tabla 2: Escenarios para la ejecución de pruebas de carga - Caso 1

Escenarios	**A**	**B**	**C**	**Detalle**
Número de hilos	50	100	400	Representa la cantidad de usuarios que interactúan con el *endpoint*.
Periodo de subida [s]	10	10	10	Tiempo en segundos en el cual todos los hilos se inician de manera gradual hasta alcanzar el número total configurado.
Contador de bucle	2	2	2	Número de iteraciones realizadas por cada hilo durante la prueba.

Estos servicios utilizan la librería `RabbitMQ.Client` para conectarse al *message broker*. Se incorporó un mecanismo de reintento exponencial para gestionar posibles fallos de conexión con RabbitMQ.

OrderController: En este componente se eliminó el código asociado a la implementación síncrona anterior y se introdujo un nuevo mecanismo para la creación y envío de mensajes al *message broker*. Esta refactorización le permite actuar como el emisor de eventos, enviando los detalles de las órdenes al *message broker* para que los servicios distribuidos realicen las operaciones correspondientes.

Por otro lado, se incorporó la librería `Polly` para la definición de políticas de reintento en caso de errores de conexión con RabbitMQ. Además, se configuraron las opciones `AutomaticRecoveryEnabled` y `NetworkRecoveryInterval`, las cuales permiten la recuperación automática en caso de fallos de red.

Finalmente, se revisaron las implementaciones para asegurar que las conexiones con RabbitMQ se cierren adecuadamente una vez completadas las tareas.

Contenedores: Todos los componentes de la solución fueron contenerizados mediante Docker. Cada servicio cuenta con su propio `Dockerfile` y para la orquestación de los contenedores se utilizó un archivo `docker-compose.yml` permitiendo desplegar el sistema completo de forma eficiente y reproducible en distintos entornos.

Monitoreo: Se implementó Serilog en cada uno de los *Worker Services*, lo que permite capturar logs detallados sobre el comportamiento de los servicios. Para centralizar y procesar estos registros, se incorporó una instancia de SEQ con configuraciones estándar, lo que facilitó la observación y análisis en tiempo real del sistema durante su operación.

3.4 Resultados y discusión

Una vez finalizada la etapa de implementación se ha obtenido dos sistemas. El primero es el sistema monolítico síncrono original (MONO1) y el segundo es el sistema implementado con una arquitectura distribuida basada en eventos (EDA) resultado de la transformación. Las características técnicas de cada sistema se presentan en la Tabla 1.

Para la evaluación del rendimiento del sistema, se ha seleccionado la herramienta Apache JMeter. Por medio de esta herramienta se ejecutaron pruebas de carga en 3 escenarios (A, B y C) cuyas características se presentan en la Tabla 2.

Luego de ejecutar las pruebas se recopilaron los resultados de uso de recursos de *hardware*, los cuales

Tabla 3: Uso de CPU y RAM - MONO1 vs EDA

Escenario	$\overline{CPU}$[1]	**CPU**$^{2}_{min}$	**CPU**$^{3}_{max}$	σ^{4}_{CPU}	$\overline{RAM}$[5]	**RAM**$^{6}_{min}$	**RAM**$^{7}_{max}$	σ^{8}_{RAM}	T^{9}_{est}
MONO1 A	1.76	0.01	16.49	3.32	41.48	21.68	49.00	7.50	1:18
MONO1 B	1.51	0	33.35	4.41	53.85	23.39	67.75	11.71	2:33
MONO1 C	2.39	0.01	41.37	5.68	67.99	23.77	83.42	12.15	3:35
EDA A	3.90	0.99	23.76	23.57	391.49	354.36	398.77	10.70	0:09
EDA B	5.57	0.84	82.34	13.13	393.81	352.37	408.64	12.22	0:09
EDA C	3.90	0.98	80.08	9.89	402.15	398.13	410.93	9.89	0:10

[1] $\overline{CPU}$: Uso CPU promedio [%]
[2] CPU$_{min}$: Uso CPU mínimo [%]
[3] CPU$_{max}$: Uso CPU máximo [%]
[4] σ_{CPU}: Desviación estándar del uso CPU [%]
[5] $\overline{RAM}$: Uso RAM promedio [MB]
[6] RAM$_{min}$: Uso RAM mínimo [MB]
[7] RAM$_{max}$: Uso RAM máximo [MB]
[8] σ_{RAM}: Desviación estándar del uso RAM [MB]
[9] T_{est}: Tiempo de estabilización [min]

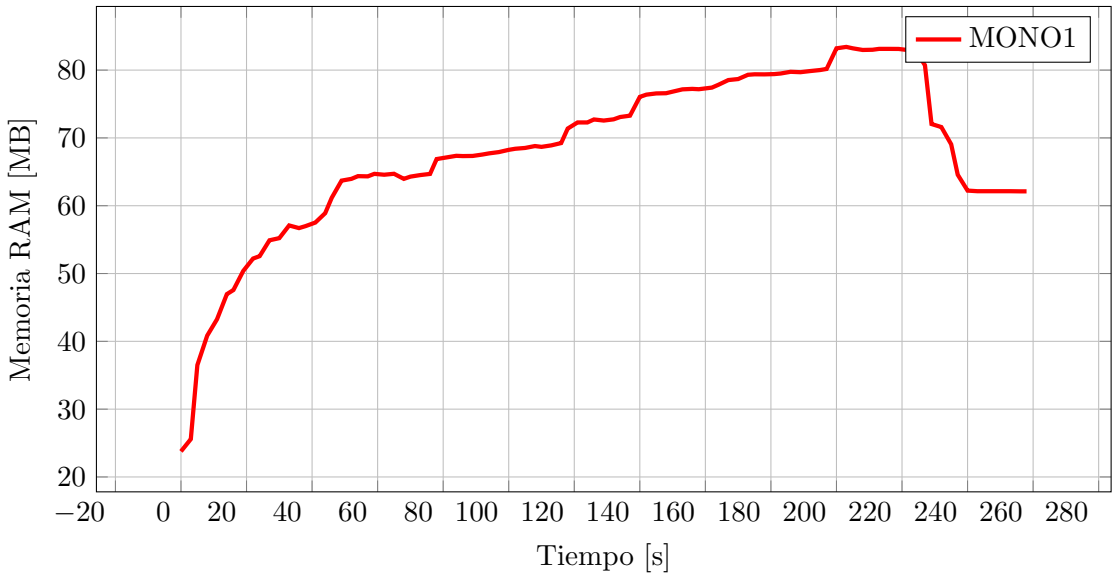

Figura 2: Uso de memoria RAM vs tiempo para MONO1 - Escenario C

se presentan en la Tabla 3

Con respecto al uso de CPU, se evidencia que el porcentaje promedio de EDA es ligeramente mayor a MONO1, sin embargo, los tiempos de estabilización de MONO1 son considerablemente más altos que los de EDA para el mismo escenario, lo que significa que MONO1 requiere una mayor cantidad de tiempo para entregar una respuesta. Además, se puede observar que la variación de carga entre los escenarios A, B y C afecta mucho más al tiempo de estabilización del sistema MONO1 en comparación con el sistema EDA.

Con respecto al uso RAM se evidencia que EDA requiere una mayor cantidad de memoria en comparación con MONO1, lo cual se debe al hecho de que EDA cuenta con seis componentes activos mientras que MONO1 tan solo tiene uno. Adicionalmente, se puede apreciar en la Figura 2 como el sistema MONO1 experimenta mayores fluctuaciones de consumo de memoria durante la prueba en comparación con EDA en la Figura 3.

De igual manera, los resultados de las pruebas de carga para los tres escenarios mencionados anteriormente se resumen en la Tabla 4. En este caso es evidente que el tiempo de respuesta promedio es muy significativo en el sistema MONO1 comparado con los tiempos del sistema EDA. En este punto es importante destacar que el tiempo de respuesta promedio en MONO1 va desde casi los 20$[s]$ hasta los 90$[s]$, lo que claramente afectará la experiencia de uso bajo estas condiciones, a diferencia de EDA que

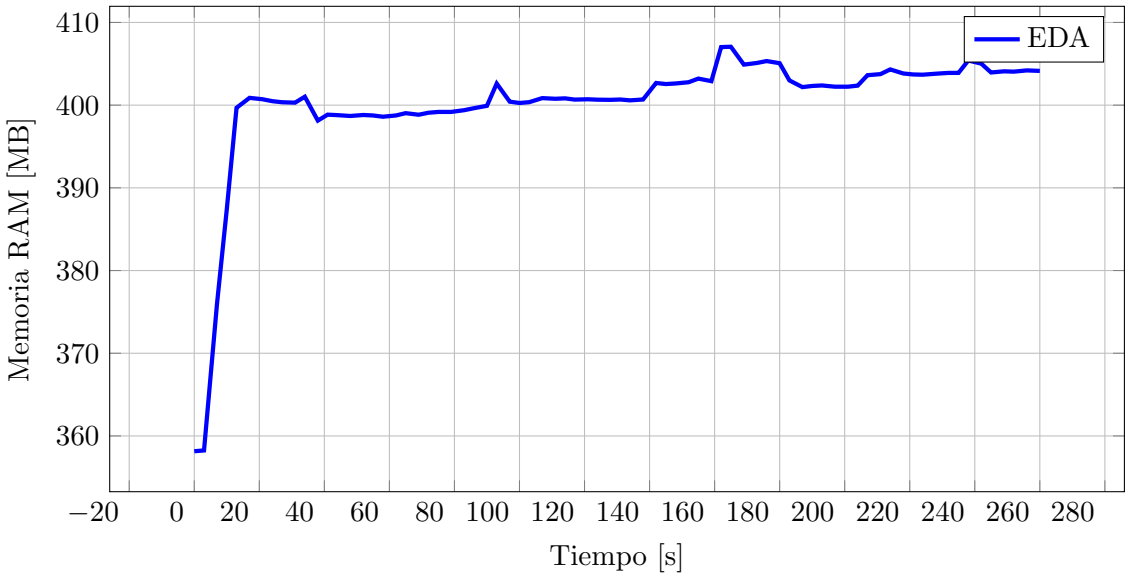

Figura 3: Uso de memoria RAM vs tiempo para EDA - Escenario C

Tabla 4: Resultados de pruebas de carga - MONO1 vs EDA

Escenario	$\bar{t}$[1]	**Mín**[2]	**Máx**[3]	σ[4]	**%Err**[5]	**Thrp**[6]	**Rx**[7]	**Tx**[8]
MONO1 A	20024	13829	28447	3609.61	0	1.9	0.33	0.48
MONO1 B	54315	12979	99340	21704.13	0	3.1	0.55	0.80
MONO1 C	93355	42609	150098	27415.57	0	3.7	0.66	0.96
EDA A	9	2	336	35.42	0	10.2	2.01	2.60
EDA B	15	1	483	67.69	0	41.1	8.11	10.48
EDA C	7	1	418	34.63	0	80.2	15.82	20.44

[1] $\bar{t}$: Tiempo de respuesta promedio [ms]
[2] Mín: Tiempo mínimo de respuesta [ms]
[3] Máx: Tiempo máximo de respuesta [ms]
[4] σ: Desviación estándar de t [ms]
[5] %Err: Porcentaje de error
[6] Thrp: Throughput [Transacciones/s]
[7] Rx: Datos recibidos [KB/s]
[8] Tx: Datos enviados [KB/s]

presenta tiempos promedio no mayores a los 15$[ms]$.

Por otro lado, el *throughput* del sistema MONO1 es bajo en comparación con EDA que crece conforme la demanda de cada escenario. Adicionalmente, ninguno de los dos sistemas presentó errores durante la ejecución de las pruebas, pero considerando los altos tiempos de respuesta promedio en MONO1, es muy probable que, en un escenario real, esto pueda ser interpretado como un error por parte de los sistemas o servicios consumidores.

En retrospectiva, el cambio más notable entre el sistema MONO1 y EDA es la adopción de un enfoque asíncrono en el procesamiento de órdenes. En la práctica, esto elimina la necesidad de que el cliente espere en línea la confirmación completa de su operación, permitiendo que las transacciones se ejecuten en segundo plano mientras el cliente es notificado al finalizar el proceso.

En cuanto al rendimiento del sistema, el cambio más significativo se observó en la drástica reducción de los tiempos de respuesta promedio y el aumento del *throughput* en EDA en comparación con MONO1, lo que destaca las ventajas de una arquitectura orientada a eventos en escenarios de alta demanda, como los aquí descritos. Si bien las pruebas mostraron que EDA requiere más recursos de *hardware* que MONO1, esta diferencia es compensada con una mayor capacidad de escalabilidad y mejor rendimiento en escenarios de alta demanda por parte de EDA, por lo que es importante considerar esta última característica al momento de optar por uno u otro enfoque.

4 Caso 2: Agendamiento de citas médicas

4.1 Análisis

4.1.1 Descripción del caso

La segunda prueba de concepto aborda el caso de un sistema monolítico desarrollado en Java que forma parte de una plataforma de agendamiento de citas médicas y que sigue los principios SOA. Este sistema está compuesto por varios controladores responsables de gestionar la información relacionada con médicos, horarios, especialidades y pagos. Este análisis se enfoca en la gestión de horarios, realizada por el controlador `AppointmentController`, el cual implementa los siguientes métodos:

- `saveSchedule`: Permite registrar las citas médicas de acuerdo con la disponibilidad de los médicos.
- `getScheduleById`: Permite obtener una cita médica en particular a partir de su identificador único.
- `getAllSchedule`: Permite obtener todas las citas médicas disponibles y es accesible tanto para los sistemas internos como para los usuarios finales.
- `updateSchedule`: Permite actualizar la información de la cita médica y se utiliza principalmente para actualizar su estado de disponibilidad.

Todas estas operaciones se ejecutan a través de una instancia `CrudRepository` e internamente hace uso de una base de datos H2 usando las API JDBC estándar.

4.1.2 Problemas detectados

Uno de los principales problemas de la aplicación radica en la gestión del acceso a la base de datos. La base de datos tiene un número limitado de conexiones concurrentes y existe un desbalance en la carga entre las operaciones de lectura y escritura. Por un lado, las operaciones de inserción y actualización son frecuentes pero moderadas, mientras que, al mismo tiempo, hay una carga constante y elevada de consultas debido a que los sistemas y usuarios finales deben verificar la disponibilidad de citas antes de seleccionar una para su agendamiento. Esto provoca que, en escenarios de alta demanda o en temporadas pico, se presenten problemas de latencia elevada y bloqueos en la base de datos.

Además, se contempla que a futuro será necesario mantener un registro de auditoría que permita consultar el estado de las citas médicas y conservar un historial de los cambios realizados, lo cual supone un riesgo debido a las condiciones actuales del sistema monolítico.

4.2 Diseño

Considerando los problemas descritos en el apartado anterior, se decidió focalizar los esfuerzos en aplicar mejoras al controlador `AppointmentController`. En este caso la estrategia aplicada para desacoplar las operaciones de lectura y escritura fue la implementación de un enfoque CQRS (*Command Query Responsibility Segregation*) combinado con *Event Sourcing*.

En la implementación original, el desbalance entre las operaciones de lectura y escritura es una característica derivada de la naturaleza del sistema, por lo que era necesario contemplar este comportamiento en el desacoplamiento del sistema y al mismo tiempo contemplar que las nuevas funcionalidades de auditoría no comprometan su rendimiento.

Por lo mencionado anteriormente la aplicación fue desacoplada en los siguientes componentes:

- `CommandController`: Este componente asume las responsabilidades de las operaciones de escritura. Internamente hace uso del servicio `ScheduleCommandService` el cual permite depositar comandos en el `CommandBus`.
- `Aggregate`: Es el responsable de recuperar los comandos del `CommandBus` haciendo uso de un *handler* para procesar los eventos `CreateScheduleItemCommand` y `UpdateScheduleItemCommand`. Estos *handlers* permiten el almacenamiento de los eventos en el *Event Store* y también permiten la actualización del estado del objeto por medio de los `EventSourcingHandlers`.

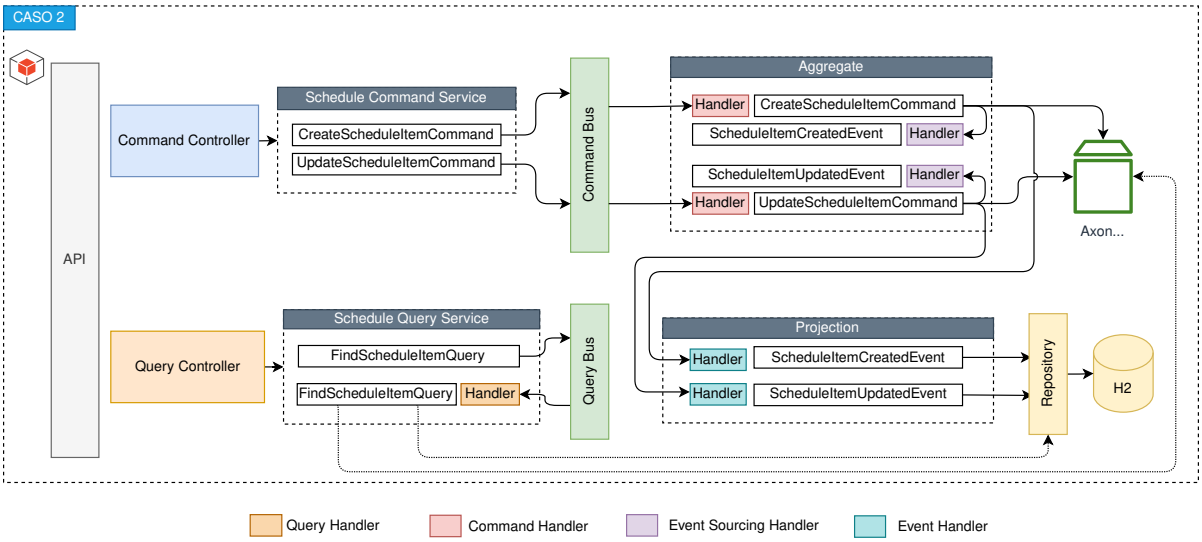

Figura 4: Descomposición del sistema monolítico con CQRS y *Event Sourcing*

- `QueryController`: Este componente asume las responsabilidades de las operaciones de lectura. Hace uso del servicio `ScheduleQueryService` para depositar los mensajes en el `QueryBus`. Dependiendo del tipo de consulta este puede redireccionarlas hacia `Repository` o hacia el *Event Store* reduciendo la carga de consultas sobre la base de datos. Internamente `GetScheduleByIdQuery` permite obtener una cita médica por su identificador, pero adicionalmente permite obtener todo el historial de eventos de esta cita hasta una fecha límite si esta se envía en la consulta.
- `Projection`: Se encarga de capturar los eventos generados en `Aggregate` y actualizar los registros en la base de datos. Por ejemplo, cuando se genera el evento `ScheduleCreatedEvent` la proyección actualizará el repositorio con una nueva entrada.
- `Event Bus`: Actúa como un gestor central para comandos y consultas.
- `Event Store`: Este componente se encarga del almacenamiento de los eventos y permite reconstruir el estado de las citas desde cero para fines de auditoría.

Para emular el comportamiento de un sistema de base de datos con los problemas descritos en etapa de análisis, se ha implementado en el monolito un mecanismo en el repositorio que genera un error de conexión aleatorio en escenarios de alta demanda. Este mecanismo fue heredado en la nueva implementación.

En la Figura 4 se muestra como estos componentes se relacionan entre sí y cuáles son los eventos asociados.

4.3 Desarrollo

En esta fase, se desarrollaron los componentes definidos en la etapa de diseño, utilizando Java y Spring Boot como principales tecnologías. Adicionalmente, se seleccionó a Axon como Framework para este desarrollo ya que cuenta con soporte para la implementación de CQRS y *Event Sourcing*. A continuación, se destacan los aspectos más relevantes de la implementación:

Controladores y servicios: Los controladores fueron implementados como `RestControllers` de Spring, inyectando los servicios mediante anotaciones `@Autowired`. Los servicios, `ScheduleCommandService` y `ScheduleQueryService`, interactúan con el sistema a través de `CommandGateway` y `QueryGateway`, que son interfaces provistas por Axon para el envío de comandos y consultas a los buses respectivos.

Aggregate y Projection: Para la implementación del agregado se extendió la clase `ScheduleItem` y adicionalmente, mediante la anotación `@Aggregate`, se marcó a esta clase como un agregado que será gestionado por Axon. Con el método `scheduleItemSnapshotTriggerDefinition`, un `@Bean` definido en la clase principal del proyecto, se configuró en veinte la cantidad de eventos para la ejecución del *trigger*

Tabla 5: Características de los sistemas - Caso 2

	MONO2	***CQRS***
Lenguaje	Java 17	Java 17
Framework	Spring Framework	Spring Framework Axon Framework
Número de archivos	13	26
Líneas de código efectivas	367	685
Comentarios	53	69
Líneas en blanco	77	182
Total de líneas	**497**	**936**

de generación *snapshots*, las cuales son imágenes parciales del estado del agregado en un punto específico en el tiempo.

Dentro de `Aggregate`, los *handlers* encargados de gestionar los comandos, hacen uso del método `AggregateLifecycle.apply()` y de esta forma los eventos son aplicados y propagados al sistema para que otros componentes puedan reaccionar a los mismos. Por ejemplo, cuando se emite un comando `CreateScheduleItemCommand`, el `Aggregate` genera un evento correspondiente que es almacenado y procesado.

Event Store y Event Bus: Para el almacenamiento y distribución de eventos, se utilizó Axon Server en modo *standalone*. Por un lado, su capacidad de actuar como *Event Store* permitió almacenar los eventos que se generan y los pone a disposición para su recuperación posterior (por ejemplo, para reconstruir el estado de los agregados). Por otro lado, como *Event Bus* permite distribuir los eventos a los suscriptores, en este caso los agregados, servicios y proyecciones que los necesitan.

Persistencia y Repositorios: El sistema originalmente hacía uso de `CrudRepository` para la persistencia de datos, pero en esta nueva versión fue sustituido por `JPARepository`, lo que añadió capacidades adicionales de consulta y manejo de entidades. Se mantuvo la base de datos en H2 para el almacenamiento de las proyecciones. Además, se integró Lombok en el desarrollo, lo que simplificó considerablemente la cantidad de código necesario al generar automáticamente métodos como `getters`, `setters` y constructores, reduciendo la cantidad de código *boilerplate* y mejorando la mantenibilidad.

Contenedores: Tanto la nueva API como la instancia de Axon Server fueron contenerizados. Se ha utilizado `docker-compose.yml` para facilitar el despliegue del sistema completo.

Monitoreo: Axon Server ofrece capacidades de monitoreo integradas, proporcionando visibilidad sobre la actividad del *Event Store* y el *Event Bus*. Esto facilita el seguimiento de la salud del sistema, permitiendo detectar posibles cuellos de botella o problemas de rendimiento en tiempo real.

4.4 Resultados y discusión

Una vez finalizada la etapa de implementación se ha obtenido dos sistemas. El primero es el sistema monolítico original (MONO2). El segundo es el sistema implementado con una arquitectura distribuida basada en eventos (CQRS) resultado de la transformación. Las características técnicas de cada sistema se presentan en la Tabla 5.

Para las evaluaciones de rendimiento, se ha utilizado la herramienta Apache JMeter, con la cual se ejecutaron pruebas de carga en 3 escenarios (A, B y C) cuyas características se presentan en la Tabla 6.

Luego de ejecutar las pruebas se recopilaron los resultados de uso de recursos de *hardware*, los cuales se presentan en la Tabla 7.

En cuanto al uso de memoria RAM, se observa un aumento significativo en el promedio en CQRS con respecto a MONO2. Este incremento puede atribuirse a la naturaleza de CQRS, que requiere el manejo de múltiples componentes adicionales, como los buses de comandos y consultas, y las proyecciones en tiempo real, lo que incrementa la carga de memoria.

Respecto al uso de CPU, CQRS alcanzó picos que superaron el 100 %, lo que indica que en determinados momentos se hizo uso de más de un núcleo para procesar las operaciones concurrentes. Por otro lado, en la Figura 5 se puede observar que, aunque el consumo de CPU en MONO2 fue menor, este mantuvo un nivel alto por un periodo de tiempo relativamente prolongado. En contraste, la Figura 6 nos muestra cómo el comportamiento del uso de CPU en CQRS se caracteriza por ráfagas intermitentes,

Tabla 6: Escenarios para la ejecución de pruebas de carga - Caso 2

Escenarios	**A**	**B**	**C**	**Detalle**
Número de hilos	25	100	250	Representa la cantidad de usuarios que interactúan con el *endpoint*.
Periodo de subida [s]	10	10	10	Tiempo en segundos en el cual todos los hilos se inician de manera gradual hasta alcanzar el número total configurado.
Contador de bucle	2	2	2	Número de iteraciones realizadas por cada hilo durante la prueba.

Tabla 7: Uso de CPU y RAM - MONO2 vs CQRS

Escenario	$\overline{CPU}$[1]	**CPU**$^2_{min}$	**CPU**$^3_{max}$	σ^4_{CPU}	$\overline{RAM}$[5]	**RAM**$^6_{min}$	**RAM**$^7_{max}$	σ^8_{RAM}	T^9_{est}
MONO2 A	3.62	0.26	17.10	6.13	410.79	404.60	421.50	7.08	0:14
MONO2 B	6.49	0.28	51.94	15.17	483.85	462.30	489.80	9.84	0:57
MONO2 C	5.32	0.31	71.66	15.99	483.33	420.00	552.70	54.90	1:17
CQRS A	25.16	2.83	73.84	30.10	1762.32	1715.90	1778.70	26.65	0:09
CQRS B	30.82	3.90	323.61	76.37	1783.82	1489.20	1817.80	91.05	0:09
CQRS C	58.32	2.51	410.60	111.02	1566.01	1279.50	1640.00	116.99	0:40

[1] $\overline{CPU}$: Uso CPU promedio [%]
[2] CPU_{min}: Uso CPU mínimo [%]
[3] CPU_{max}: Uso CPU máximo [%]
[4] σ_{CPU}: Desviación estándar del uso CPU [%]
[5] $\overline{RAM}$: Uso RAM promedio [MB]
[6] RAM_{min}: Uso RAM mínimo [MB]
[7] RAM_{max}: Uso RAM máximo [MB]
[8] σ_{RAM}: Desviación estándar del uso RAM [MB]
[9] T_{est}: Tiempo de estabilización [min]

Tabla 8: Resultados de pruebas de carga - MONO2 vs CQRS

Escenario	$\bar{t}$[1]	**Mín**[2]	**Máx**[3]	σ[4]	**%Err**[5]	**Thrp**[6]	**Rx**[7]	**Tx**[8]
MONO2 A	965	47	1871	525.37	0	10.2	19.24	2.72
MONO2 B	966	17	1913	547.96	0.17	34.6	240.34	9.22
MONO2 C	974	15	1915	534.91	0.33	82.5	1412.57	21.98
CQRS A	19	17	147	18.61	0	15.7	18.48	1.76
CQRS B	15	5	164	12.79	0	60.2	239.41	15.36
CQRS C	19	7	21577	6205.41	0	37.2	359.27	9.49

[1] $\bar{t}$: Tiempo de respuesta promedio [ms]
[2] Mín: Tiempo mínimo de respuesta [ms]
[3] Máx: Tiempo máximo de respuesta [ms]
[4] σ: Desviación estándar de t [ms]
[5] %Err: Porcentaje de error
[6] Thrp: Throughput [Transacciones/s]
[7] Rx: Datos recibidos [KB/s]
[8] Tx: Datos enviados [KB/s]

reflejando picos altos solo en momentos específicos del procesamiento.

Los resultados de las pruebas de carga para los mismos escenarios indicados previamente se resumen en la Tabla 8.

En términos de rendimiento, se observa que, a medida que se incrementa la carga sobre el sistema MONO2, comienzan a registrarse porcentajes de error significativos en los escenarios B y C lo que puede impactar negativamente en el desempeño del sistema en un entorno de producción con estas características.

Por otro lado, aunque el sistema basado en CQRS experimenta una disminución en su rendimiento bajo el escenario de mayor demanda, este mantiene los porcentajes de error en cero, lo que mejora

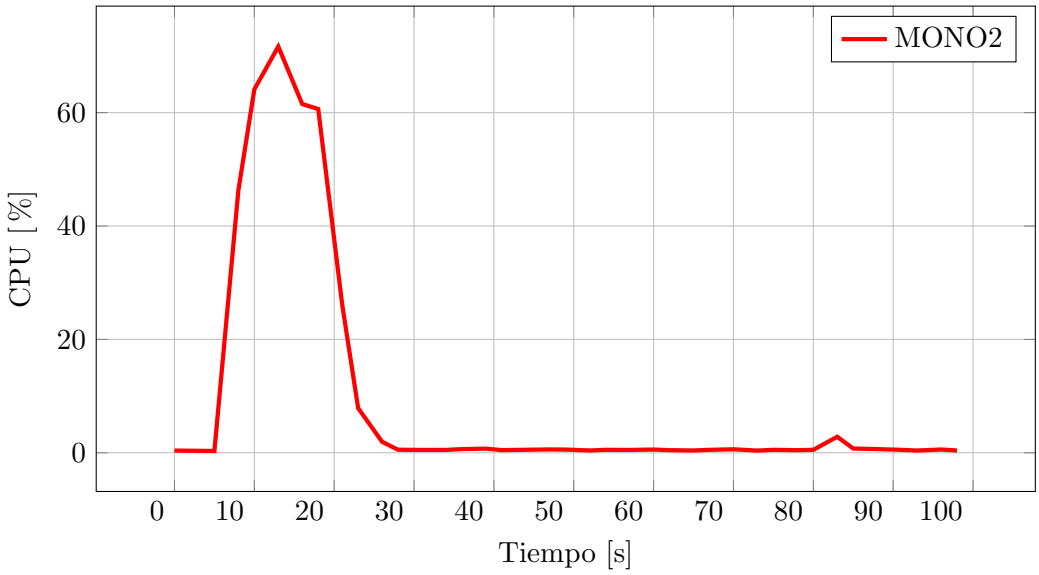

Figura 5: Uso de CPU vs tiempo para MONO2 - Escenario C

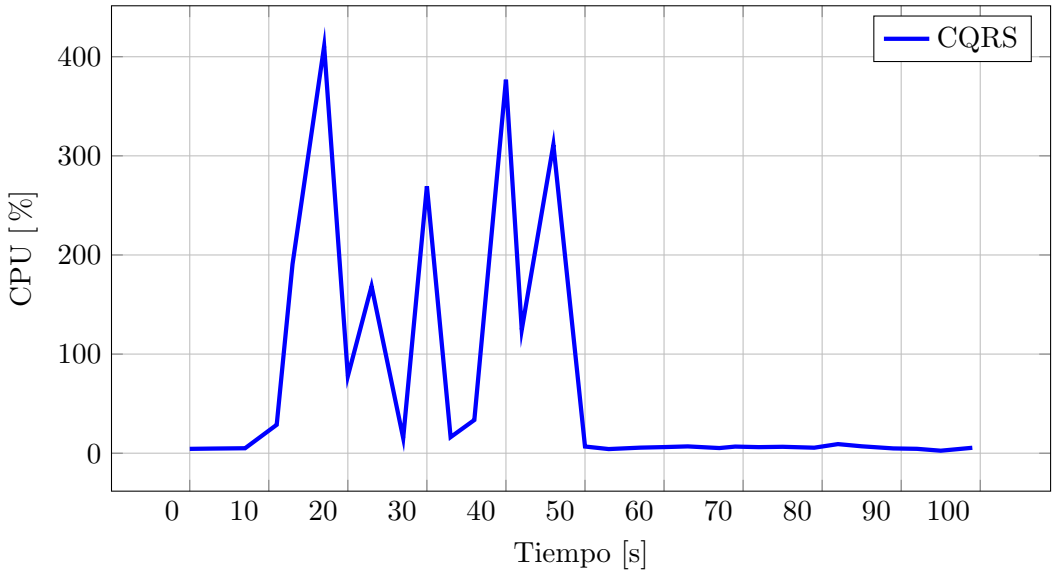

Figura 6: Uso de CPU vs tiempo para CQRS - Escenario C

la confiabilidad del sistema respecto a MONO2. Este comportamiento es particularmente destacable considerando que ambos sistemas están utilizando el mismo repositorio, el cual emula defectos de acceso descritos en la fase de diseño.

En retrospectiva, se observa que la implementación monolítica original aún podría ser una alternativa válida siempre y cuando se cuente con evidencia suficiente de que, incluso con la implementación de nuevas funcionalidades, no se superará los niveles de demanda descritos en el escenario A, de lo contrario una implementación como CQRS es la mejor alternativa.

5 Caso 3: Gestión de viajes

5.1 Análisis

5.1.1 Descripción del caso

La tercera prueba de concepto aborda el caso de un sistema monolítico destinado a la gestión de operaciones de una agencia de viajes. Como parte del modelo de negocio la empresa actúa como intermediaria entre los usuarios y proveedores externos para la búsqueda, reserva y pagos servicios relacionados con hoteles, vehículos y vuelos.

Para soportar estas operaciones la empresa cuenta con un sistema cuyo *backend*, desarrollado en .NET, implementa una única API monolítica la cual expone los siguientes métodos:

- Reservaciones únicas: Consta de tres métodos que permiten gestionar las reservas de hoteles, vehículos y vuelos de forma independiente.
- Reservaciones combinadas: Consta de cuatro métodos que permiten la reserva de paquetes combinados, como vuelos más hotel o vehículo más hotel y que necesitan ser coordinadas entre múltiples proveedores.
- Consultas: Consta de seis métodos que permiten consultar el resultado de las operaciones de reserva de forma individual.
- Pagos: Consta de dos métodos que permiten obtener el detalle de los pagos realizados generados por las operaciones de reserva o sus combinaciones.

Internamente, todos los métodos del sistema dependen de llamadas síncronas a los servicios de terceros para completar las transacciones, como los servicios de pago y los servicios específicos de los proveedores.

5.1.2 Problemas detectados

Como fue descrito anteriormente, se puede notar que todas las funcionalidades están integradas en una única API, por lo que cualquier cambio en un servicio individual (por ejemplo, actualizaciones en la lógica de reserva de hoteles) requiere el despliegue completo de todo el sistema, lo que aumenta la complejidad y el riesgo de introducción de errores en producción.

Por otro lado, durante las temporadas de alta demanda, el sistema experimenta tiempos de espera elevados debido a la heterogeneidad de los servicios externos de los proveedores y sus tiempos de respuesta, así como la naturaleza síncrona de las invocaciones realizadas a los servicios externos.

Esta latencia adicional afecta la experiencia del usuario y reduce la eficiencia del sistema. Otro aspecto para destacar es que actualmente la empresa está en proceso de expansión, y las áreas de negocio (hoteles, vehículos, vuelos) crecen a ritmos diferentes. Sin embargo, la arquitectura monolítica limita la capacidad de escalar de manera independiente cada área.

Finalmente, todo el sistema centraliza la información en una única base de datos, lo que crea un cuello de botella en términos de rendimiento y disponibilidad. Además, cualquier fallo en esta base comprometerá a todas las áreas.

5.2 Diseño

Con base en las características y problemas identificados en la sección anterior, se decidió utilizar el patrón *API Gateway* como estrategia principal para la descomposición progresiva del sistema monolítico, esto con la finalidad de que los cambios en los componentes internos detrás del *API Gateway* sean transparentes

para los consumidores, facilitando la migración paulatina sin interrumpir las operaciones del sistema. Internamente se ha optado por implementar una arquitectura basada en microservicios principalmente para brindar una mayor independencia a las áreas involucradas. A continuación, se detallan los principales componentes involucrados en la implementación:

- ***API Gateway***: Es el punto de entrada del sistema. Inicialmente, su función principal consiste en enrutar las solicitudes directamente hacia los métodos del monolito. Sin embargo, conforme avanza el proceso de desarrollo, la configuración de este componente cambia progresivamente para redirigir las solicitudes hacia los nuevos microservicios.
- **Servicio orquestador**: Este elemento se encarga de analizar los mensajes recibidos y descomponerlos si es necesario. Dependiendo del tipo de mensaje el orquestador tiene la capacidad de enviarlo hacia el sistema de mensajería o redireccionarlo directamente al microservicio si se detecta que puede ser procesado de forma autónoma.
- **Sistema de mensajería**: Se encarga de la gestión y enrutamiento de los mensajes recibidos hacia los consumidores correspondientes. Este componente es clave para manejar la comunicación asíncrona entre microservicios, desacoplando las dependencias temporales entre ellos. Para garantizar su disponibilidad estará compuesto por un clúster distribuido y para facilitar el monitoreo cuenta con un sistema de administración.
- **Microservicios**: Se han diseñado cuatro microservicios especializados, cada uno enfocado en un dominio funcional específico: Hoteles, Vuelos, Vehículos y Pagos. Cada microservicio es autónomo y responsable de su propia lógica de negocio para facilitar el escalado independiente según las necesidades de la empresa.
- **Sistema de descubrimiento de servicios**: Su función principal es registrar, localizar y monitorizar los servicios disponibles en el sistema. Además, proporciona información en tiempo real sobre el estado y la salud de los servicios registrados.
- **Funciones de pago**: Son integraciones externas con proveedores de servicios de pago. Estas funciones permiten abstraer la lógica de procesamiento de pagos fuera del sistema central, lo que facilita la posibilidad de cambiar o agregar proveedores sin afectar la lógica de negocio.

Para comprender como los componentes descritos anteriormente y representados en la Figura 7 interactúan entre sí, consideremos como ejemplo el flujo para un proceso de reserva completo. Este flujo inicia cuando el mensaje llega al *API Gateway*, que lo enruta hacia el Orquestador; este lo analiza, descompone y lo envía al Sistema de Mensajería o directamente a un microservicio según corresponda. En el Sistema de Mensajería los mensajes son recuperados por los microservicios consumidores, los cuales procesan los mensajes de manera autónoma, y en el caso de operaciones combinadas se devuelve el mensaje al Sistema de Mensajería para completar su procesamiento por medio del microservicio Agrupador de pagos. En cualquier caso, los microservicios que procesan los mensajes invocan a las funciones de pago para finalizar su transacción.

5.3 Desarrollo

En esta fase se desarrollaron los componentes diseñados previamente, utilizando herramientas y tecnologías cuyas características, criterios de selección y detalles de implementación se describen a continuación.

API Gateway: Se seleccionó a Kong como *API Gateway*, destacándose por su capacidad de escalabilidad y balanceo de carga. La versión utilizada fue la 3.7.1, obtenida desde el Docker Registry oficial. Kong cumple el rol de proxy reverso, habilitando la configuración declarativa a través de archivos YAML y también el modo `DB-less`, con el uso de variables de entorno desde `docker-compose`. Se configuró un puerto para administración y se definieron rutas para la redirección hacia el Orquestador.

Sistema de Mensajería: Se desplegó un *clúster* de Apache Kafka con tres nodos, usando el modo `Kraft` ya que, para el nivel de complejidad de este sistema, no es necesario el uso de ZooKeeper. La configuración de los nodos incluyó propiedades como `controller.quorum.voters`, `listeners`, y `process.roles`, donde los nodos eligen al *broker controller* a través de un proceso de consenso en el quorum. Para la gestión de los mensajes del sistema se crearon los siguientes tópicos:

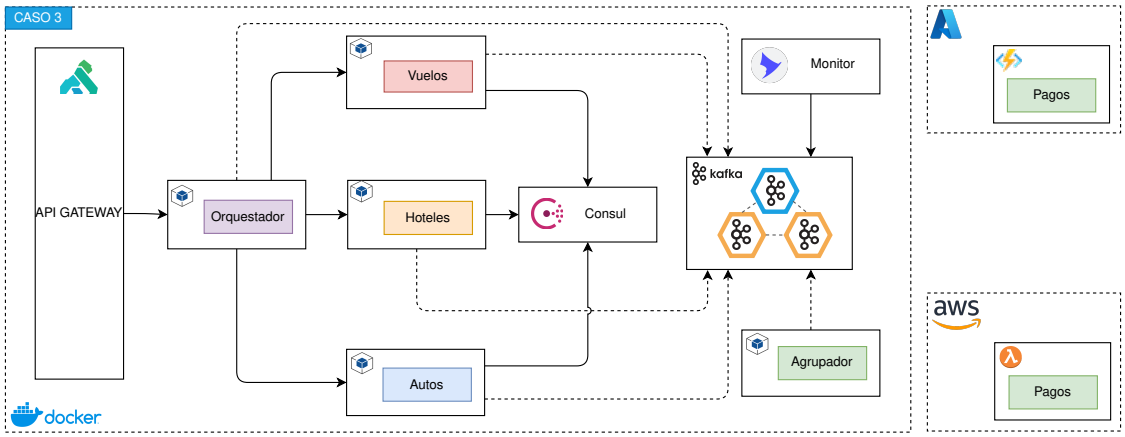

Figura 7: Descomposición del sistema monolítico en microservicios

- `FlightReservationTopic`
- `HotelReservationTopic`
- `CarReservationTopic`
- `PaymentGrouperTopic`

Por otro lado, para monitorear el *clúster*, se utilizó UI for Apache Kafka (v0.7.2), facilitando la observabilidad del estado del *cluster*, tópicos y mensajes.

Sistema de descubrimiento de servicios: Se implementó Consul (v1.10.0) en modo *Service Discovery* y *Key-Value Store* para gestionar la configuración dinámica de los microservicios. La configuración del servidor incluye el nodo de Consul con interfaces HTTP expuestas y monitoreo activado. Cada microservicio se registra en Consul y se verifica periódicamente con *healthchecks*.

Microservicios: Los microservicios, implementados en .NET 8, utilizan un patrón productor-consumidor con `Confluent.Kafka` para integrarse con Kafka. Además, cada microservicio cuenta con una base de datos local SQLite propia, administrada con Entity Framework y ejecutando migraciones para su mantenimiento. Los microservicios incluyen un cliente para Consul que les permite registrarse automáticamente en el *Service Discovery*. Para la gestión de configuraciones dinámicas, se usa `Winton.Extensions.Configuration.Consul`, lo que permite cambiar en caliente entre sistemas de pago mediante Consul `Key-Value`. Los microservicios desarrollados bajo este esquema fueron: `service-flight`, `service-hotel`, `service-car` y `service-paymentgrouper`.

Orquestador: Es un servicio REST implementado en .NET 8 que actúa como intermediario entre Kong y el *cluster* de Kafka. Internamente, implementa la lógica necesaria para descomponer los mensajes recibidos y reenviarlos según las definiciones del diseño.

Función de Pagos: Se desarrollaron dos funciones *serverless* para procesar pagos. La primera fue una Azure Function en .NET, integrada con GitHub Actions para mejorar la velocidad de despliegue mediante un pipeline de CI/CD y la segunda una Lambda Function en .NET con despliegue manual sobre AWS.

Todos los componentes descritos anteriormente se implementaron en contenedores mediante Docker Compose.

5.4 Resultados y discusión

Una vez finalizada la etapa de implementación se ha obtenido dos sistemas. El primero es el sistema monolítico original (MONO3) y el segundo es el sistema implementado con una arquitectura distribuida basada en microservicios (MICRO) resultado de la transformación. Las características técnicas de cada sistema se presentan en la Tabla 9.

Para las evaluaciones de rendimiento, se ha utilizado la herramienta Apache JMeter, con la cual se ejecutaron pruebas de carga en 3 escenarios (A, B y C) cuyas características se presentan en la Tabla 10.

Luego de ejecutar las pruebas se recopilaron los resultados de uso de recursos de *hardware*, los cuales se presentan en la Tabla 11.

Tabla 9: Características de los sistemas - Caso 3

	MONO3	***MICRO***
Lenguaje	C# 12	C# 12
Framework	.Net 8	.Net 8
Número de archivos	19	56
Líneas de código efectivas	656	1778
Comentarios	21	419
Líneas en blanco	85	385
Total de líneas	**762**	**2582**

Tabla 10: Escenarios para la ejecución de pruebas de carga - Caso 3

Escenarios	**A**	**B**	**C**	**Detalle**
Número de hilos	50	100	200	Representa la cantidad de usuarios que interactúan con el *endpoint*.
Periodo de subida [s]	10	10	10	Tiempo en segundos en el cual todos los hilos se inician de manera gradual hasta alcanzar el número total configurado.
Contador de bucle	2	2	2	Número de iteraciones realizadas por cada hilo durante la prueba.

Tabla 11: Uso de CPU y RAM - MONO3 vs MICRO

Escenario	$\overline{CPU}$[1]	$\mathbf{CPU}^{2}_{min}$	$\mathbf{CPU}^{3}_{max}$	σ^{4}_{CPU}	$\overline{RAM}$[5]	$\mathbf{RAM}^{6}_{min}$	$\mathbf{RAM}^{7}_{max}$	σ^{8}_{RAM}	T^{9}_{est}
MONO3 A	7.47	0.01	31.84	11.36	96.99	58.01	113.80	21.61	0:20
MONO3 B	4.00	0.01	57.07	11.25	115.79	41.92	142.30	23.98	0:21
MONO3 C	3.30	0.01	73.62	11.03	155.68	41.88	197.70	31.10	0:45
MICRO A	23.27	11.20	74.44	14.08	2866.12	2835.41	2881.74	14.29	0:09
MICRO B	22.75	9.92	97.64	18.36	2819.64	2777.61	2840.79	14.92	0:10
MICRO C	33.42	9.81	470.74	72.18	3034.61	2882.96	3075.60	45.14	0:10

[1] $\overline{CPU}$: Uso CPU promedio [%]
[2] CPU_{min}: Uso CPU mínimo [%]
[3] CPU_{max}: Uso CPU máximo [%]
[4] σ_{CPU}: Desviación estándar del uso CPU [%]
[5] $\overline{RAM}$: Uso RAM promedio [MB]
[6] RAM_{min}: Uso RAM mínimo [MB]
[7] RAM_{max}: Uso RAM máximo [MB]
[8] σ_{RAM}: Desviación estándar del uso RAM [MB]
[9] T_{est}: Tiempo de estabilización [min]

Respecto al uso de memoria RAM, se puede apreciar en la Figura 8 que el sistema MICRO consume una mayor cantidad respecto a su contraparte. Esto puede explicarse principalmente porque MONO3 solo cuenta con un único componente su arquitectura, mientras que MICRO cuenta con al menos doce.

En cuanto al uso de CPU, se observa un importante aumento en el promedio en MICRO con respecto a MONO3. Este incremento de igual forma puede atribuirse a la cantidad de componentes que se encuentran activos, que individualmente pueden mantener un uso relativamente bajo, pero en conjunto representa una cantidad considerable. Adicionalmente se observa que en un determinado momento MICRO hizo uso de más de un núcleo para atender sus operaciones. En la Figura 9, se evidencia a pesar de la diferencia en los niveles de consumo el comportamiento de MICRO es más estable.

Los resultados de las pruebas de carga para los mismos escenarios indicados previamente se resumen en la Tabla 12.

En términos de rendimiento, se observa que, al incrementar la carga en el sistema MONO3, su

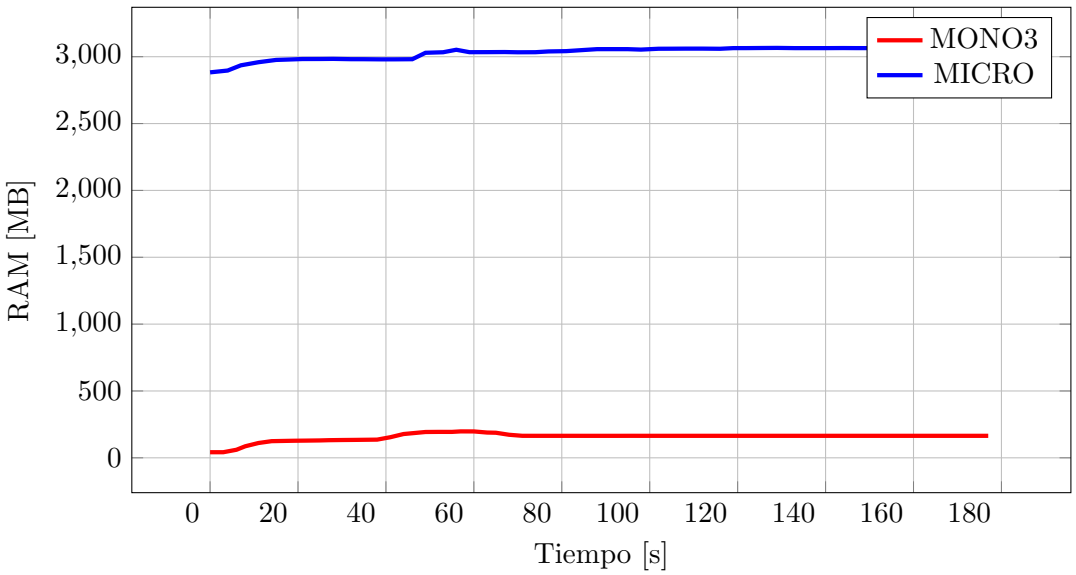

Figura 8: Uso de RAM vs tiempo para MONO3 y MICRO - Escenario C

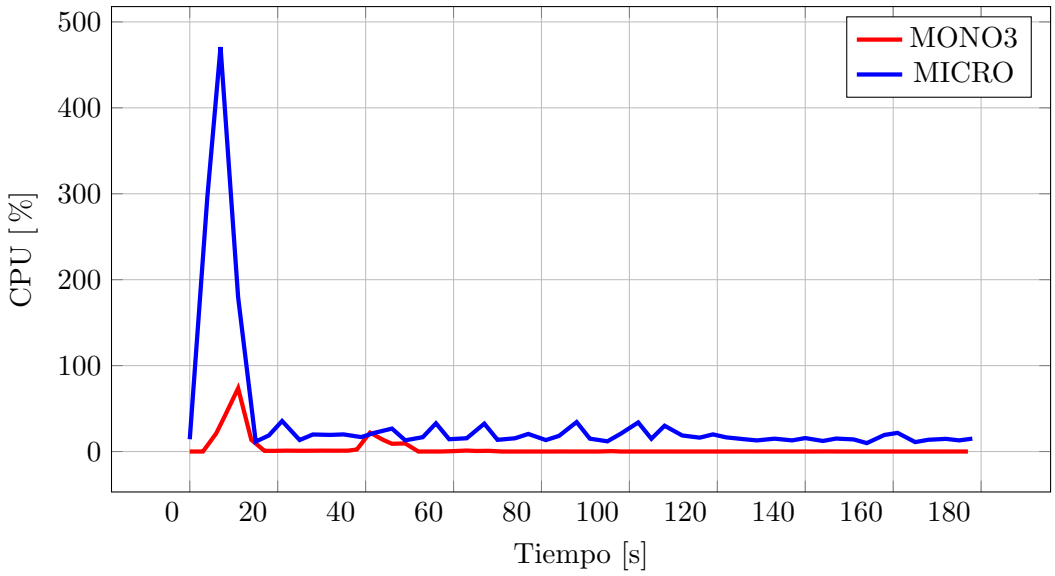

Figura 9: Uso de CPU vs tiempo para MONO3 y MICRO - Escenario C

Tabla 12: Resultados de pruebas de carga - MONO3 vs MICRO

Escenario	$\bar{t}$[1]	**Mín**[2]	**Máx**[3]	σ[4]	**%Err**[5]	**Thrp**[6]	**Rx**[7]	**Tx**[8]
MONO3 A	4597	1595	6012	1016.35	0	4.8	0.95	3.54
MONO3 B	4789	1467	6296	1045.27	0	9.4	1.85	6.92
MONO3 C	17948	2453	35465	12503.30	0	8.8	1.72	6.45
MICRO A	22	14	52	7.28	0	10.2	3.36	7.45
MICRO B	21	14	46	5.30	0	20.1	6.63	14.72
MICRO C*	25	13	77	9.41	0	40.0	13.21	29.30

[1] $\bar{t}$: Tiempo de respuesta promedio [ms]
[2] Mín: Tiempo mínimo de respuesta [ms]
[3] Máx: Tiempo máximo de respuesta [ms]
[4] σ: Desviación estándar de t [ms]
[5] %Err: Porcentaje de error
[6] Thrp: Throughput [Transacciones/s]
[7] Rx: Datos recibidos [KB/s]
[8] Tx: Datos enviados [KB/s]

rendimiento tiende a disminuir ligeramente. Si bien en todos los casos las tasas de error se mantienen en cero, es evidente que los tiempos de respuesta de MONO3 se incrementan considerablemente, alcanzando niveles que, en un entorno de producción, evidentemente representarán un problema.

Por otro lado, destaca el hecho de que MICRO, incluso en el escenario de mayor demanda, logró mantener niveles muy aceptables, tanto en rendimiento como en tiempos de respuesta. En este punto es importante mencionar que para el escenario MICRO C* originalmente el tiempo promedio de MICRO fue de $44[ms]$, el cual también es aceptable. Sin embargo, durante las pruebas de carga se pudo determinar que en alta demanda el orquestador en muy pocas ocasiones (menos del 1 % de los casos) generaba un cuello de botella. Para mitigar este efecto se agregó una segunda instancia del orquestador y se configuró un `upstream` en Kong que actúa como balanceador de carga entre las dos instancias. Con esto se consiguió reducir el tiempo de respuesta promedio MICRO en el escenario C casi a los niveles de A y B.

En retrospectiva, si bien se observa que el sistema MICRO requiere más recursos a nivel de *hardware* para su funcionamiento, este lo compensa con un desempeño considerablemente mayor que el sistema MONO3, principalmente en escenarios de alta demanda, por lo cual resulta ser la mejor alternativa para el caso analizado.

6 Conclusiones y Recomendaciones

El presente trabajo aborda el proceso de transformación de aplicaciones monolíticas las cuales, por medio de la selección e implementación de varias estrategias, ha sido posible evolucionarlas hacía arquitecturas distribuidas desacopladas.

La transformación del sistema monolítico síncrono a una arquitectura distribuida basada en eventos descrita en la Sección 3 ha mostrado importantes mejoras, ya que fue posible comprobar durante su evaluación como la adopción de EDA y un enfoque asíncrono reduce significativamente los tiempos de espera experimentados por los usuarios en un entorno de alto tráfico, el cual era un problema importante evidenciado en la arquitectura monolítica original. Para este mismo caso, el uso de RabbitMQ como *message broker* fue fundamental para desacoplar los servicios clave, facilitando la comunicación asíncrona entre ellos. Por otro lado, la adopción de un enfoque reactivo, donde los servicios se mantienen escuchando eventos provenientes de las colas fue posible gracias al uso de *Worker Services* de .NET.

Al mismo tiempo, es altamente recomendable que, al implementar una arquitectura basada en eventos, se apliquen estrategias para mejorar la confiabilidad del sistema. En este caso, la introducción de reintentos exponenciales en los *Workers Services*, gestionar los fallos de comunicación en `OrderController`, y configuraciones de alta disponibilidad en RabbitMQ, aseguraron un rendimiento robusto incluso en situaciones de alta demanda.

Para el segundo caso descrito en la Sección 4, la introducción de CQRS y *Event Sourcing* mejoró la estabilidad y confiabilidad del sistema frente a fallos y condiciones adversas. Aunque en la prueba de concepto se observó una disminución en el *throughput* en el escenario de mayor demanda, la nueva arquitectura lo compensa con una mayor resiliencia, manteniendo en cero los porcentajes de error. Esta característica muestra que CQRS con *Event Sourcing* es una alternativa viable para la implementación

de sistemas donde la confiabilidad es prioritaria.

En este mismo caso, el uso de tecnologías como Spring Boot, Axon Framework y Axon Server han sido fundamentales ya que permitieron agregar nuevas características al sistema, como la capacidad de manejar comandos y eventos de manera desacoplada. Particularmente, el uso de Axon ha facilitado la introducción de un modelo basado en eventos, permitiendo que las proyecciones o vistas de lectura se actualicen de manera asíncrona y confiable incluso en escenarios adversos. Por otro lado, aunque esta prueba de concepto no representa una implementación estricta de *Domain Driven Design* (DDD), las bases establecidas por el uso de Axon Framework dejan abierta la posibilidad de evolucionar gradualmente hacia un diseño más alineado con dichos principios.

Si bien existen algunas ventajas con respecto a CQRS y *Event Sourcing*, es importante entender que el sistema tiene un modelo de consistencia eventual. Esto significa que después de que se envían los eventos, las proyecciones y las vistas de consulta pueden no estar inmediatamente actualizadas en todas las instancias, pero eventualmente reflejarán el estado actual de los eventos. Para el sistema descrito en la Sección 4 esto no representa un problema ya que toda cita es reconfirmada en el proceso de agendamiento, sin embargo, esto no necesariamente será igual en otros escenarios por lo que se recomienda tener muy en cuenta este factor antes de considerar su adopción.

Con respecto a las herramientas y componentes utilizados en esta segunda prueba de concepto con CQRS se ha utilizado una instancia de Axon Server en modo *Standalone*, lo cual ha sido suficiente para el escenario descrito; sin embargo, para sistemas que deban manejar una demanda significativamente mayor, se recomienda configurar Axon Server en modo *Cluster*, lo que proporcionará mayor disponibilidad, resiliencia y capacidad de escalado horizontal.

Para el proceso de transformación descrito en la Sección 5, la adopción del patrón *API Gateway* como estrategia para la descomposición gradual del sistema monolítico a una arquitectura orientada a microservicios permitió garantizar la continuidad del servicio y al mismo tiempo mantener una experiencia uniforme para los consumidores. Adicionalmente, la integración de un sistema de mensajería robusto como lo fue el *cluster* Apache Kafka facilitan el desacoplamiento de las funcionalidades, al tiempo que asegura la escalabilidad, flexibilidad y resiliencia del sistema.

Así mismo, la incorporación de componentes como Kong, Apache Kafka y Consul pone de manifiesto la importancia de una comunicación eficiente y robusta en arquitecturas de componentes altamente desacoplados, como es el caso de los microservicios. La capacidad de estos componentes para procesar mensajes de forma asíncrona, descubrir dinámicamente nuevos servicios y balancear la carga de manera autónoma es crucial para garantizar la flexibilidad, escalabilidad y resiliencia de la solución. Estas características no solo permiten que cada microservicio gestione su dominio de negocio de manera eficiente, sino que también facilitan el mantenimiento y la evolución del sistema a largo plazo.

Durante la etapa de pruebas, el proceso implementado para mejorar el tiempo de respuesta del sistema demostró cómo, en arquitecturas distribuidas con componentes altamente desacoplados, el escalamiento de recursos es una alternativa viable y de fácil ejecución. Específicamente en el caso descrito, el escalamiento horizontal del componente Orquestador se llevó a cabo en pocos minutos, lo que resultó en una mejora importante en el tiempo promedio de respuesta, reduciéndolo aproximadamente en un 25 %. Este resultado destaca la capacidad de adaptación y eficiencia de las arquitecturas desacopladas frente a incrementos en la carga del sistema.

Del mismo modo y con base en la experiencia obtenida durante la implementación de este último caso, se recomienda tener precaución al utilizar el patrón *API Gateway*, ya que este puede convertirse en un único punto de falla del sistema. Para mitigar este riesgo, es recomendable dividir el *API Gateway* en múltiples *gateways* más pequeños, por ejemplo, orientados a diferentes límites de negocio (*business boundaries*). Esta estrategia, además de mejorar la disponibilidad y escalabilidad, también permite una mayor flexibilidad y desacoplamiento entre los distintos dominios de negocio, facilitando el mantenimiento y la evolución del sistema.

Por otro lado, y desde un punto de vista más general, se pudo observar también que el proceso de contenerización de todos los componentes utilizando Docker garantizó la capacidad de desplegar el sistema de manera sencilla y consistente en múltiples entornos, demostrando así su aporte en factores como la flexibilidad y escalabilidad en el despliegue. De igual forma, a través del uso de marcos de software como Spring Boot, .NET, las pruebas de concepto no solo demuestran los beneficios técnicos, sino también cómo estas tecnologías facilitan la implementación de sistemas escalables, confiables y tolerantes a fallos.

Cabe destacar también que durante la implementación de los tres casos, los procesos de análisis del estado actual del sistema se enfocaron en la identificación de eventos en tiempo de ejecución que generaban

las mayores cargas o latencias, el análisis de la estructura interna de los monolitos y la identificación de modelos delimitados por las fronteras de negocio. Estas actividades están estrechamente relacionadas con los principios de *Dynamic Analysis*, *Static Analysis* y *Model-Driven Development*, técnicas clave que permitieron descomponer los sistemas monolíticos y orientar su transformación hacia arquitecturas distribuidas y desacopladas.

Finalmente es importante mencionar también que, en cualquier proceso de transformación o evolución de sistemas, es fundamental implementar herramientas de monitoreo y observabilidad para mantener un control preciso sobre su estado. En los casos analizados, el uso de herramientas como Seq, Axon Server y Consul proporcionó información importante para comprender el comportamiento del sistema durante su implementación y pruebas, contribuyendo al éxito del proceso de transformación.

Referencias

A. Aggarwal and V. Singh. Migration aspects from monolith to distributed systems using software code build and deployment time and latency perspective. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 22(4):854, Aug. 2024. ISSN 2302-9293, 1693-6930. doi: 10.12928/telkomnika.v22i4.25655. URL http://telkomnika.uad.ac.id/index.php/TELKOMNIKA/article/view/25655.

H. Cabane and K. Farias. On the impact of event-driven architecture on performance: An exploratory study. *Future Gener. Comput. Syst.*, 153(C):52–69, May 2024. ISSN 0167-739X. doi: 10.1016/j.future.2023.10.021. URL https://doi.org/10.1016/j.future.2023.10.021.

T. Cerny, M. J. Donahoo, and J. Pechanec. Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, pages 228–235, Krakow Poland, Sept. 2017. ACM. ISBN 9781450350273. doi: 10.1145/3129676.3129682. URL https://dl.acm.org/doi/10.1145/3129676.3129682.

T. Clark and B. S. Barn. A common basis for modelling service-oriented and event-driven architecture. In *Proceedings of the 5th India Software Engineering Conference*, pages 23–32, Kanpur India, Feb. 2012. ACM. ISBN 9781450311427. doi: 10.1145/2134254.2134258. URL https://dl.acm.org/doi/10.1145/2134254.2134258.

N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow, Apr. 2017. URL http://arxiv.org/abs/1606.04036. arXiv:1606.04036 [cs] version: 4.

Y. Gao, G. Casale, and R. Singhal. Performance Modeling of Distributed Data Processing in Microservice Applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, page 3687265, Aug. 2024. ISSN 2376-3639, 2376-3647. doi: 10.1145/3687265. URL https://dl.acm.org/doi/10.1145/3687265.

Y. M. Lau, C. M. Koh, and L. Jiang. Teaching Software Development for Real-World Problems using a Microservice-Based Collaborative Problem-Solving Approach. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, pages 22–33, Lisbon Portugal, Apr. 2024. ACM. ISBN 9798400704987. doi: 10.1145/3639474.3640064. URL https://dl.acm.org/doi/10.1145/3639474.3640064.

Lisa J. Kirby, Evelien Boerstra, Zachary J.C. Anderson, and Julia Rubin. Weighing the Evidence: On Relationship Types in Microservice Extraction. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 358–368, Madrid, Spain, May 2021. IEEE. ISBN 9781665414036. doi: 10.1109/ICPC52881.2021.00041. URL https://ieeexplore.ieee.org/document/9463006/.

Z. Long. Improvement and Implementation of a High Performance CQRS Architecture. In *2017 International Conference on Robots & Intelligent System (ICRIS)*, pages 170–173, Huai An City, China, Oct. 2017. IEEE. ISBN 9781538612279. doi: 10.1109/ICRIS.2017.49. URL http://ieeexplore.ieee.org/document/8101372/.

J. A. Oliveira, M. Vargas, and R. Rodrigues. SOA Reuse: Systematic Literature Review Updating and Research Directions. In *Proceedings of the XIV Brazilian Symposium on Information Systems*, pages 1–8, Caxias do Sul Brazil, June 2018. ACM. ISBN 9781450365598. doi: 10.1145/3229345.3229419. URL https://dl.acm.org/doi/10.1145/3229345.3229419.

M. Overeem, M. Spoor, and S. Jansen. The dark side of event sourcing: Managing data conversion. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 193–204, Klagenfurt, Austria, Feb. 2017. IEEE. ISBN 9781509055012. doi: 10.1109/SANER.2017.7884621. URL http://ieeexplore.ieee.org/document/7884621/.

V. Sharma, H. K. Saxena, and A. K. Singh. Docker for Multi-containers Web Application. In *2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, pages 589–592, Bangalore, India, Mar. 2020. IEEE. ISBN 9781728141671. doi: 10.1109/ICIMIA48430.2020.9074925. URL https://ieeexplore.ieee.org/document/9074925/.

N. Trabelsi, C. Politowski, and G. E. Boussaidi. Event Driven Architecture: An Exploratory Study on The Gap between Academia and Industry. In *2023 IEEE/ACM 5th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, pages 25–32, Melbourne, Australia, May 2023. IEEE. ISBN 9798350301885. doi: 10.1109/SERP4IoT59158.2023.00010. URL https://ieeexplore.ieee.org/document/10190452/.

Valentina Lenarduzzi, Francesco Lomio, Nyyti Saarimäki, and Davide Taibi. Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software*, 169:110710, Nov. 2020. ISSN 01641212. doi: 10.1016/j.jss.2020.110710. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121220301539.

M. Younas, D. N. A. Jawawi, A. K. Mahmood, M. N. Ahmad, M. U. Sarwar, and M. Y. Idris. Agile Software Development Using Cloud Computing: A Case Study. *IEEE Access*, 8:4475–4484, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2962257. URL https://ieeexplore.ieee.org/document/8943168/.

7 Biografía

Héctor David Andrade Unusungo. El autor es estudiante del programa de Maestría de Software. El obtuvo su título de Ingeniero en Electrónica y Redes de Información en el año 2018 por la Escuela Politécnica Nacional. Actualmente se desempeña como Ingeniero de desarrollo y Líder Técnico en la empresa Bayteq Cia. Ltda. handradeu@est.ups.edu.ec