

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

SEDE VIÑA DEL MAR - JOSÉ MIGUEL CARRERA

MODERNIZACIÓN DE UN SISTEMA MONOLÍTICO SIMULADO MEDIANTE ARQUITECTURA DE MICROSERVICIOS CON COMUNICACIÓN ASINCRÓNICA BASADA EN PUB/SUB ORIENTADO A ENTORNOS CLOUD-NATIVE

Trabajo de Titulación para optar al Título

de Técnico Universitario en INFORMÁTICA

Alumno: Jorge Antonio Rubén Martínez Martorell

Profesora Guía: Lais San Martín Navarro

2026

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: MODERNIZACIÓN DE UN SISTEMA MONOLÍTICO SIMULADO MEDIANTE ARQUITECTURA DE MICROSERVICIOS CON COMUNICACIÓN ASINCRÓNICA BASADA EN PUB/SUB ORIENTADO A ENTORNOS CLOUD-NATIVE

Nombre del candidato(a): Jorge Martínez Martorell

Carrera / Grado: Técnico Universitario en Informática

Campus: Sede Viña del Mar **Departamento:** Departamento de Electrotecnia e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Lais San Martín Navarro, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 19-octubre-2026 **Firma:** Lais San Martín N

Estudiante o Candidato(a):

Fecha: 20-octubre-2026 **Firma:**

Resumen

El presente Trabajo de Título aborda la modernización de un sistema monolítico simulado, de reservas de vuelos, mediante el diseño e implementación de una arquitectura de microservicios con comunicación asíncrona, basada en el patrón Pub/Sub, desplegada en un entorno cloud-native local. La motivación nace de las limitaciones técnicas observadas en arquitecturas monolíticas cuando crecen en tamaño y frecuencia de cambio: alto acoplamiento interno, complejidad en los despliegues, escalabilidad horizontal restringida y costos operativos crecientes asociados al escalado global de la aplicación. A partir de un caso de estudio que simula el sistema monolítico de una aerolínea —centrado en el flujo de reserva, la confirmación de pago y la notificación al pasajero—, el trabajo modela el sistema base, identifica sus puntos críticos de acoplamiento y los traduce en requerimientos funcionales, no funcionales, actores y casos de uso. Sobre esta base, se define una arquitectura distribuida que descompone las responsabilidades del monolito en servicios autónomos —un módulo principal de reservas, un microservicio de Notificaciones y un microservicio de Recomendaciones—, comunicados mediante eventos y soportados por el patrón database-per-service con PostgreSQL. La metodología es iterativa e incremental, y la evaluación técnica se construye en torno a escenarios de prueba y métricas vinculadas a latencia, throughput, escalabilidad horizontal, resiliencia, idempotencia y observabilidad, instrumentadas con Prometheus y Grafana. El análisis comparativo se presenta mediante tablas que contrastan ambos enfoques bajo criterios cualitativos y operativos. Las conclusiones evidencian que el enfoque de microservicios con comunicación asíncrona ofrece mejores condiciones para el desacoplamiento, la evolución independiente de componentes y la escalabilidad selectiva, sin que ello constituya una mejora automática o universal. Su adopción introduce nueva complejidad operativa, exige consistencia eventual entre servicios y depende de un soporte de infraestructura adecuado, condiciones que deben evaluarse caso a caso para que la modernización resulte efectivamente conveniente.

KEYWORDS: MICROSERVICIOS · PUB/SUB · MONOLITO · ARQUITECTURA EVENT-DRIVEN · CLOUD-NATIVE · POSTGRESQL.

ÍNDICE

Introducción 1
Capítulo 1. Análisis de la situación actual y sistema objetivo de modernización 3
1.1 Transformación digital y contexto 3
1.2 Contexto de desarrollo del proyecto 3
1.3 Caracterización del sistema monolítico 4
1.3.1 Implicaciones, ventajas y limitaciones del enfoque monolítico 4
1.4 Caracterización del sistema de microservicios 5
1.5 Caso de estudio: sistema de reservas de vuelos 6
1.6 Problemas detectados y justificación de la modernización 8
1.7 Síntesis del capítulo 9
Capítulo 2. Aspectos relevantes del diseño 10
2.1 Objetivos 10
2.2 Propuesta arquitectónica 11
2.3 Flujo principal del sistema 13
2.4 Requerimientos funcionales 15
2.5 Actores y casos de uso 17
2.6 Diagrama de secuencia 19
2.7 Requerimientos no funcionales 20
2.8 Información que manejará el sistema 23
2.9 Entidades de datos y persistencia 23
2.9.1 Tablas principales del sistema 28
2.9.1.1 Tabla principal: reserva 29
2.9.1.2 Tabla principal: vuelo 30
2.9.1.3 Tabla principal: pasajero 31
2.9.1.4 Tabla principal: asiento 32
2.9.1.5 Tabla principal: pago 33
2.9.1.6 Tabla principal: notificación 34
2.9.1.7 Tabla principal: recomendación 35
2.10 Metodología, alcance y resultados esperados 36
2.11 Recurso computacional, software y vistas del sistema 36
2.11.1 Configuración general del recurso computacional 37
2.11.2 Diagrama de menús 38
2.11.3 Vistas de usuario 39
2.12 Síntesis del capítulo 43
Capítulo 3. Evaluación técnica y resultados 44
3.1 Objetivo de la evaluación 44
3.2 Escenarios de prueba y métricas 44
3.3 Análisis comparativo entre enfoques 45
3.4 Presentación e interpretación de resultados 46
3.5 Síntesis del capítulo 50

Conclusiones y observaciones **51**
Referencias bibliográficas **52**

ÍNDICE DE FIGURAS

Figura 1-1. Arquitectura actual del sistema monolítico 7
Figura 1-2. Arquitectura base del sistema modernizado 11
Figura 2-1. Flujo general del proceso de reserva y propagación de eventos 14
Figura 2-2. Casos de uso principales del sistema de reservas modernizado 18
Figura 2-3. Secuencia de procesamiento del evento reserva_confirmada 19
Figura 2-4. Modelo de datos relacional del sistema de reservas 25
Figura 2-5. Diagrama de menús del sistema 30
Figura 2-6. Pantalla de búsqueda de vuelos 31
Figura 2-7. Pantalla de selección de asiento 32
Figura 2-8. Pantalla de datos del pasajero y pago 33
Figura 2-9. Pantalla de confirmación de reserva 34
Figura 2-10. Panel de monitoreo del administrador 35

ÍNDICE DE TABLAS

Tabla 3-1. Matriz de evaluación comparativa 38
Tabla 3-2. Interpretación cualitativa de los enfoques comparados 41

Sigla y Simbología

SIGLA

ACID: Atomicity, Consistency, Isolation, Durability (Atomicidad, Consistencia, Aislamiento y Durabilidad)
API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
BFF: Backend For Frontend (Backend para el Frontend)
CI: Continuous Integration (Integración Continua)
CQRS: Command Query Responsibility Segregation (Segregación de Responsabilidades de Comando y Consulta)
CU: Caso de Uso
DLQ: Dead Letter Queue (Cola de Mensajes No Entregados)
HTTP: Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto)
OLTP: Online Transaction Processing (Procesamiento Transaccional en Línea)
p95: Percentil 95 (latencia bajo la cual responde el 95 % de las solicitudes)
p99: Percentil 99 (latencia bajo la cual responde el 99 % de las solicitudes)
Pub/Sub: Publisher/Subscriber (Publicador/Suscriptor)
REST: Representational State Transfer (Transferencia de Estado Representacional)
RF: Requerimiento Funcional
RNF: Requerimiento No Funcional
RPS: Requests Per Second (Solicitudes por Segundo)

DICCIONARIO TÉRMINOS TÉCNICOS

API composition: Patrón de agregación en que un servicio compuesto consulta a varios microservicios y combina sus respuestas en una única respuesta hacia el cliente
CQRS: Patrón arquitectónico que separa la responsabilidad de leer datos (consultas) de la de modificarlos (comandos), permitiendo construir vistas materializadas optimizadas para lectura, típicamente alimentadas por eventos

Database-per-service: Patrón de persistencia en que cada microservicio dispone de su propia base de datos independiente, sin compartir esquema ni instancia con otros servicios

Docker: Plataforma de contenedores que empaqueta software con sus dependencias

FastAPI: Framework web de alto rendimiento para construir APIs con Python

Grafana: Herramienta de visualización de métricas y construcción de tableros

K3s: Distribución ligera de Kubernetes orientada a entornos con recursos limitados

Kubernetes: Orquestador de contenedores que automatiza despliegue, escalado y operación

Minikube: Herramienta para ejecutar un clúster Kubernetes local con fines de desarrollo

Persistencia poliglota: Estrategia en que distintos microservicios adoptan motores de almacenamiento diferentes según la naturaleza de su carga de trabajo

PostgreSQL: Sistema de gestión de bases de datos relacional de código abierto

Prometheus: Sistema de monitoreo y alertas basado en métricas de series temporales

Introducción

La transformación digital se ha consolidado como uno de los principales motores de cambio en las organizaciones contemporáneas. En este escenario, empresas de distintos sectores enfrentan la necesidad de adaptar sus sistemas de información para responder a exigencias crecientes de agilidad, escalabilidad, interoperabilidad e incorporación de nuevas capacidades tecnológicas, tales como analítica avanzada, inteligencia artificial y servicios distribuidos en la nube. Sin embargo, una parte importante de estos entornos aún depende de sistemas heredados concebidos bajo arquitecturas monolíticas que, si bien resultaron adecuadas en su contexto de origen, hoy pueden transformarse en una limitación para la evolución del negocio (Bass et al., 2021).

La génesis de este trabajo se sitúa precisamente en la necesidad de comprender, en un contexto controlado, como operan los sistemas monolíticos y de qué manera pueden evolucionar hacia modelos arquitectónicos más desacoplados. En consecuencia, la presente investigación no busca sostener de manera apriorística la superioridad de un enfoque sobre otro, sino observar, desde una perspectiva técnica y aplicada, los beneficios y desafíos asociados a un proceso de modernización arquitectónica.

Diversos referentes de la ingeniería de software, entre ellos Martin Fowler y Sam Newman, han señalado que las arquitecturas monolíticas pueden dificultar la escalabilidad selectiva, el despliegue independiente y la integración con nuevas tecnologías cuando los sistemas alcanzan un mayor nivel de complejidad (Fowler, 2014; Newman, 2021). En concordancia con estas observaciones, y considerando además la experiencia profesional del autor en desarrollo backend e integración de sistemas, resulta pertinente abordar una simulación que permita estudiar las implicancias técnicas de esta transición.

En coherencia con la modalidad de titulación especial, el presente trabajo pone el acento en el análisis, el diseño y la validación técnica controlada de la solución propuesta, sin requerir una implementación productiva completa del sistema. Por ello, la investigación se orienta

principalmente a modelar el sistema de origen, identificar sus restricciones, diseñar una arquitectura modernizada y establecer mecanismos técnicos que permitan evaluar su conveniencia dentro de un entorno académico y acotado.

Sobre esta base, el proyecto consiste en la construcción conceptual y simulada de un sistema de reservas de vuelos de carácter monolítico y su posterior modernización hacia una arquitectura de microservicios con comunicación asíncrona basada en Pub/Sub, desplegada en un entorno cloud-native local. Para ello, el trabajo considera como ejes metodológicos el modelado del sistema base, la identificación de sus restricciones técnicas, el diseño de una arquitectura distribuida y la evaluación comparativa de los beneficios observables en términos de despliegue, resiliencia, trazabilidad y extensibilidad.

En consecuencia, el estudio busca aportar una mirada práctica al proceso de modernización de sistemas, reforzando la comprensión aplicada de arquitecturas modernas y proporcionando una referencia académica útil para futuros ejercicios de análisis y diseño en contextos profesionales similares.

Capítulo 1. Análisis de la situación actual y sistema objetivo de modernización

1.1 Transformación digital y contexto

El avance tecnológico y la necesidad de mejorar la competitividad han impulsado a las organizaciones a desarrollar procesos de transformación digital. Este fenómeno implica la adopción de herramientas, prácticas y paradigmas que permitan responder con mayor agilidad a un entorno cambiante. Dentro de este proceso, la modernización de los sistemas de información constituye una de las tareas más críticas, especialmente en organizaciones que aún dependen de aplicaciones heredadas construidas bajo arquitecturas monolíticas.

En este marco, el presente trabajo propone el estudio de un caso representativo de modernización tecnológica, tomando como punto de partida un sistema monolítico de reservas de vuelos. La elección de este caso permite observar, con suficiente claridad, los efectos del acoplamiento funcional, las necesidades de integración y las posibilidades de evolución hacia una arquitectura distribuida basada en eventos.

1.2 Contexto de desarrollo del proyecto

El presente trabajo no se desarrolla al interior de una empresa o institución específica, sino en el marco de un proyecto personal de carácter académico, sustentado en la experiencia profesional del autor en el ámbito del desarrollo backend, la integración de sistemas y las arquitecturas modernas de software. En este sentido, el sistema de reservas de vuelos planteado no representa una implementación corporativa real, sino un caso de estudio simulado, construido con el propósito de analizar y diseñar una estrategia de modernización arquitectónica.

Aunque no existe una organización formal asociada al proyecto, el problema abordado surge de situaciones observables en entornos empresariales reales, particularmente en organizaciones que mantienen aplicaciones monolíticas y requieren evolucionar hacia modelos más desacoplados. Por ello, el caso de estudio conserva pertinencia técnica y académica, aun cuando su desarrollo se efectúa en un entorno controlado y formativo.

1.3 Caracterización del sistema monolítico

Un sistema monolítico puede definirse como una aplicación en la que las distintas funcionalidades del negocio se encuentran integradas en un único bloque de software, compartiendo, por lo general, una misma base de código y una misma base de datos. Este enfoque se caracteriza por una fuerte cohesión estructural y por dependencias internas que facilitan la centralización del procesamiento, pero que pueden dificultar la evolución progresiva del sistema a medida que crecen sus requerimientos (Bass et al., 2021).

En sus etapas iniciales, este tipo de arquitectura ofrece ventajas relevantes, tales como simplicidad de desarrollo, menor complejidad operativa y una administración más directa de la consistencia transaccional. No obstante, en escenarios donde el sistema requiere mayor flexibilidad, escalabilidad selectiva e integración con componentes externos, dichas ventajas pueden verse compensadas por limitaciones estructurales (Fowler, 2014).

1.3.1 Implicaciones, ventajas y limitaciones del enfoque monolítico

Las arquitecturas monolíticas han sido fundamentales en la historia del desarrollo de software empresarial. Su permanencia durante décadas no es accidental, sino resultado de ventajas concretas, entre ellas la centralización de la lógica de negocio, la simplicidad de despliegue inicial y la facilidad para mantener una visión unificada del sistema.

Sin embargo, cuando la aplicación crece en tamaño, criticidad o frecuencia de cambio, comienzan a manifestarse ciertas restricciones técnicas. Entre ellas destacan la escalabilidad global, que obliga a aumentar recursos para toda la aplicación aun cuando solo un módulo concentre la demanda; la mantenibilidad dependiente de la estructura interna, que puede exigir pruebas extensivas ante cambios locales; y la dificultad para incorporar nuevos servicios o mecanismos de integración sincrónica sin aumentar el acoplamiento existente (Newman, 2021; Richards, 2015).

Estas observaciones no implican desestimar el valor del monolito como opción arquitectónica, sino contextualizar su idoneidad. En dominios acotados, estables o con baja complejidad de integración, un monolito puede seguir siendo plenamente razonable. No obstante, en escenarios donde la evolución funcional, la resiliencia y la trazabilidad adquieren relevancia, se vuelve pertinente explorar alternativas más modulares.

1.4 Caracterización del sistema de microservicios

Una arquitectura de microservicios puede definirse como un estilo en el que las distintas funcionalidades del negocio se descomponen en servicios pequeños, autónomos y desplegables de forma independiente, cada uno con su propia base de datos y comunicándose entre sí mediante protocolos ligeros (generalmente HTTP/REST o mensajería asíncrona). A diferencia del enfoque monolítico, no existe un único bloque ejecutable, sino una colección de servicios que evolucionan, escalan y se mantienen por separado bajo su propia responsabilidad funcional.

Entre las ventajas reconocidas de este estilo arquitectónico se cuentan la autonomía operativa de cada servicio, la capacidad de escalar componentes críticos sin replicar la aplicación completa, la posibilidad de adoptar tecnologías heterogéneas según la naturaleza de cada problema, y la mayor resiliencia frente a fallos parciales, ya que la caída de un servicio no detiene necesariamente al resto del sistema (Newman, 2021; Richardson, 2018).

El sistema objetivo de modernización conserva el dominio funcional del caso base, pero reorganiza ciertas responsabilidades bajo una arquitectura de microservicios. En esta etapa, cada servicio posee una responsabilidad más acotada, una persistencia propia y mecanismos de comunicación asincrónica basados en eventos publicados en Pub/Sub (Kleppmann, 2017).

Para efectos del trabajo, se consideran dos componentes clave en esta evolución: el microservicio de Notificaciones, encargado de consumir eventos de reserva confirmada y emitir mensajes al usuario; y el microservicio de Recomendaciones, responsable de procesar dichos eventos para generar sugerencias de upgrade de cabina o promociones. De esta forma, el sistema resultante permite observar un desacoplamiento funcional concreto en torno a procesos secundarios que antes podían formar parte de una única unidad ejecutable.

1.5 Caso de estudio: sistema de reservas de vuelos

El caso desarrollado en este estudio corresponde a un sistema de reservas de vuelos simulado. Su función principal es servir como base de análisis para observar, en un entorno controlado, las características del enfoque monolítico y las posibilidades de su modernización posterior. Para ello, el sistema integra funcionalidades tales como registro y autenticación de pasajeros, gestión de reservas, emisión de boletos, asignación de asientos, procesamiento de pagos simulados y envío de notificaciones.

La elección de este dominio responde a que un sistema de reservas concentra un flujo transaccional central suficientemente claro, incorpora dependencias entre entidades y permite identificar con facilidad procesos complementarios susceptibles de desacoplarse mediante eventos. Por ello, constituye un escenario adecuado para estudiar la evolución desde un enfoque centralizado hacia una arquitectura distribuida.
Aunque el caso no se asocia a una aerolínea específica del mercado, su diseño se inspira en patrones operativos comunes a las aerolíneas comerciales. Las principales funcionalidades del sistema simulado son:
- Consulta de vuelos disponibles: el pasajero busca itinerarios según origen, destino y fecha.

- Selección y reserva de asiento: el sistema bloquea un asiento del vuelo seleccionado y crea una reserva en estado pendiente.
- Confirmación de pago simulado: el usuario confirma un medio de pago ficticio, y la reserva pasa a estado confirmada, momento en que se publica el evento reserva_confirmada.
- Envío de notificación: un microservicio especializado consume el evento y emite la confirmación al pasajero (correo o canal simulado).
- Generación de recomendaciones: un microservicio independiente procesa el mismo evento y produce sugerencias de upgrade o promociones según el perfil del pasajero.

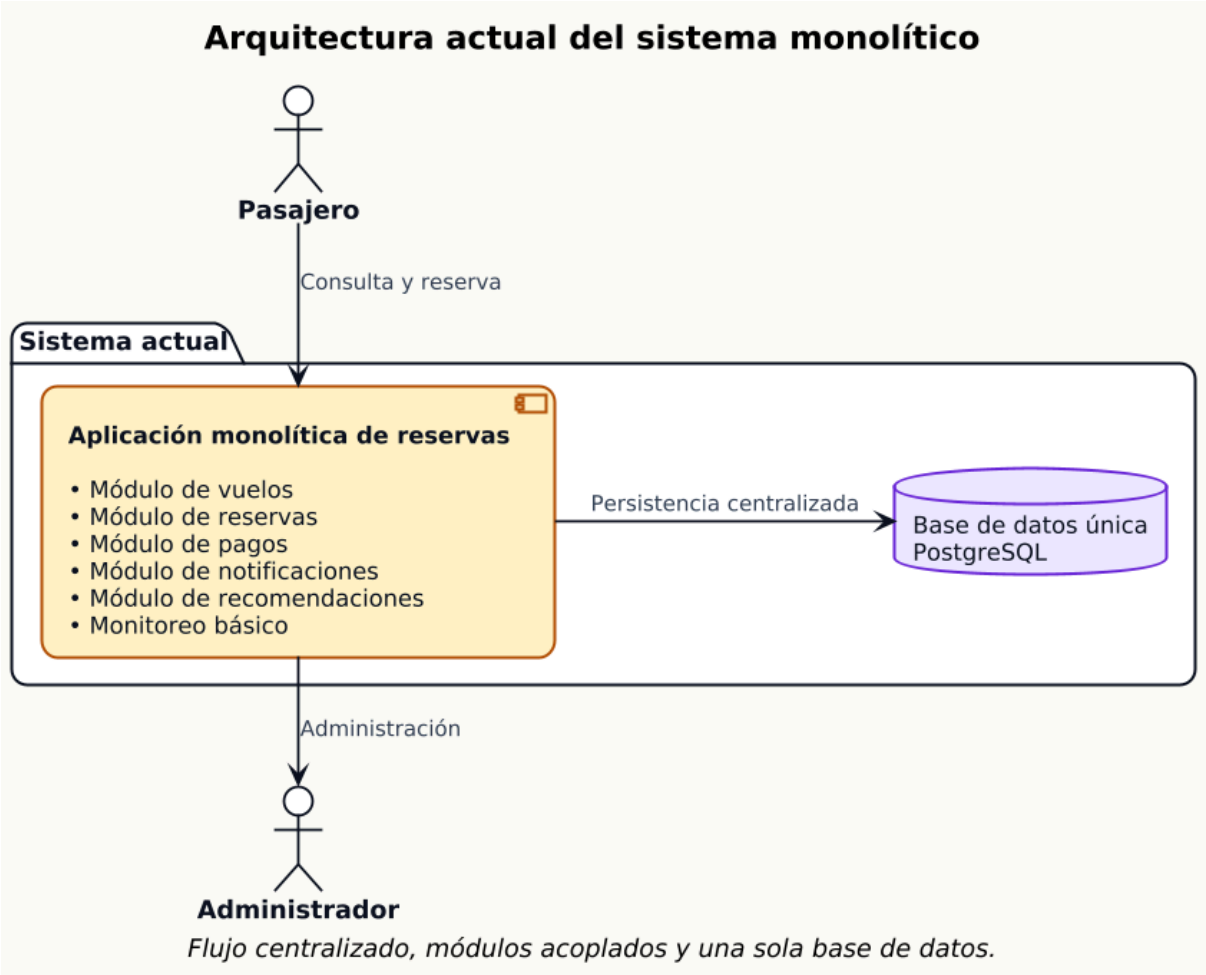


Figura 1-1. Arquitectura actual del sistema monolítico

Fuente: elaboración propia.

La Figura 1-1 representa la arquitectura actual del sistema monolítico, caracterizada por la centralización de la lógica de negocio y la persistencia en una única aplicación con una sola base de datos.

1.6 Problemas detectados y justificación de la modernización

Del análisis desarrollado se desprenden cuatro problemas principales:

- **Alto acoplamiento interno**: la incorporación de nuevas funcionalidades conlleva el riesgo de afectar otras partes del sistema, dado el fuerte acoplamiento entre módulos.
- **Complejidad de los despliegues**: cualquier cambio obliga a intervenir y validar la aplicación completa, prolongando los tiempos de salida a producción.
- **Limitación en la escalabilidad horizontal**: el aumento sostenido en la demanda de pasajeros y operaciones no puede resolverse de manera selectiva por componente.
- **Costos operativos crecientes**: el escalado global de la aplicación, en lugar del escalado específico de módulos críticos, encarece la operación.

Sobre esta base, la modernización propuesta se justifica como una estrategia que favorece la escalabilidad mediante el despliegue y crecimiento independiente de servicios, aumenta la resiliencia al aislar fallos parciales, optimiza el uso de recursos y facilita la integración con componentes tecnológicos modernos, como sistemas de mensajería, herramientas de observabilidad y servicios de recomendación (Fowler, 2014; Newman, 2021).

1.7 Síntesis del capítulo

El análisis efectuado permite concluir que el sistema monolítico de reservas de vuelos constituye un caso de estudio pertinente para examinar, desde una perspectiva técnica, las limitaciones del acoplamiento estructural y las oportunidades de evolución hacia una arquitectura más desacoplada. En consecuencia, el capítulo siguiente aborda los aspectos relevantes del diseño de la solución propuesta, con el fin de traducir esta estrategia de modernización a componentes, flujos, entidades y mecanismos de integración concretos.

Capítulo 2. Aspectos relevantes del diseño

2.1 Objetivos

Objetivo general

Simular la modernización de un sistema monolítico de reservas de vuelos mediante el diseño e implementación de una arquitectura de microservicios con comunicación asíncrona basada en Pub/Sub, desplegada en un entorno cloud-native local, generando evidencia técnica que permita comparar ambos enfoques arquitectónicos.

Objetivos específicos

1. Caracterizar el sistema monolítico base de reservas de vuelos, identificando sus puntos de acoplamiento y las limitaciones técnicas que justifican una estrategia de modernización.
2. Definir los requerimientos funcionales y no funcionales que orientan el rediseño hacia una arquitectura de microservicios, junto con los actores y casos de uso involucrados.
3. Diseñar la arquitectura distribuida del sistema objetivo, incorporando el patrón Pub/Sub para la comunicación entre servicios y el patrón database-per-service para la persistencia independiente.
4. Establecer escenarios de prueba y métricas (latencia, throughput, escalabilidad, resiliencia e idempotencia) que permitan evaluar técnicamente el comportamiento del sistema bajo cada enfoque.
5. Comparar los enfoques monolítico y de microservicios a partir de la evidencia recopilada, exponiendo ventajas, limitaciones y condiciones bajo las cuales conviene aplicar cada uno.

El patrón Pub/Sub (publicador/suscriptor) empleado en estos objetivos corresponde a un modelo de mensajería asíncrona en el que los productores de eventos (publicadores) emiten mensajes sin conocer a sus destinatarios, mientras que los consumidores (suscriptores) reciben aquellos mensajes a los que se han suscrito previamente. El bus actúa como intermediario, lo que permite que productores y consumidores evolucionen y escalen de forma independiente.

2.2 Propuesta arquitectónica

La propuesta arquitectónica desarrollada en este capítulo consiste en evolucionar el sistema monolítico de reservas hacia una arquitectura de microservicios orientada a eventos, manteniendo el dominio funcional y resolviendo las limitaciones identificadas en el sistema actual (Capítulo 1). La comparación entre ambas arquitecturas se ilustra mediante la Figura 1-1, que presenta la arquitectura actual del sistema monolítico, y la Figura 1-2, que muestra la arquitectura base del sistema modernizado.

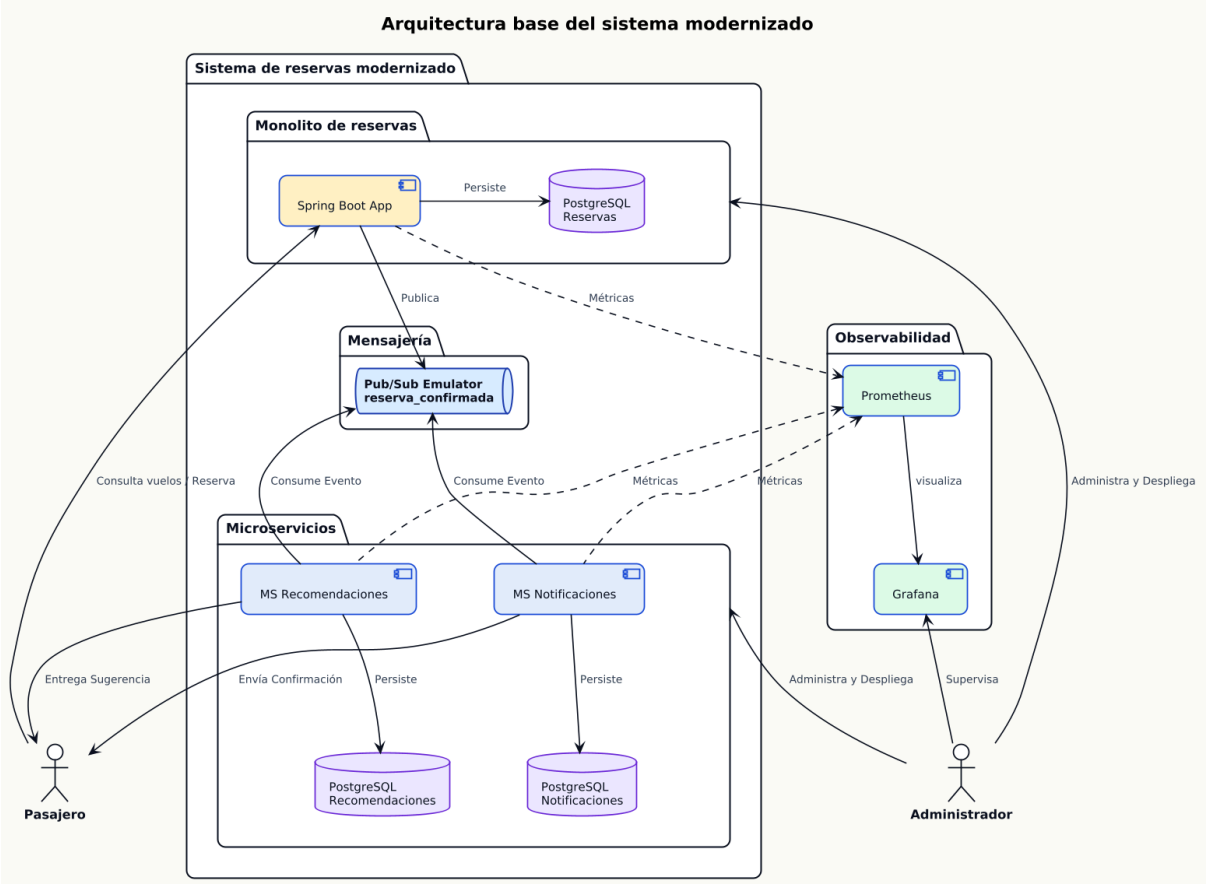


Figura 1-2. Arquitectura base del sistema modernizado

Fuente: elaboración propia.

La Figura 1-2 muestra la arquitectura base del sistema modernizado, evidenciando el desacoplamiento funcional entre el módulo principal, el bus de eventos y los servicios especializados.

En este contexto, el desacoplamiento funcional consiste en separar las responsabilidades del sistema en unidades independientes que se comunican mediante mensajes asíncronos, en lugar de invocarse directamente entre sí. Esto permite que cada componente evolucione, se despliegue y escale por separado, sin que un cambio en uno obligue a modificar o detener a los demás. En esta arquitectura, dicho desacoplamiento se logra mediante el uso del bus de eventos Pub/Sub, que actúa como intermediario entre el módulo principal de reservas y los servicios especializados de Notificaciones y Recomendaciones.

En la arquitectura propuesta, el módulo principal de reservas conserva la responsabilidad transaccional y la persistencia primaria de las reservas, mientras que los procesos secundarios (notificación al pasajero y generación de recomendaciones de upgrade) se delegan a microservicios independientes que se comunican mediante el bus de eventos Pub/Sub. Esta separación permite que cada componente evolucione y escale de manera autónoma, sin acoplamiento a nivel de persistencia ni de despliegue.

Las diferencias clave entre ambas arquitecturas son las siguientes:
- Acoplamiento: en la arquitectura monolítica, la lógica de notificaciones y recomendaciones reside dentro del mismo proceso que las reservas, lo que obliga a desplegar la aplicación completa ante cualquier modificación. En la arquitectura propuesta, cada microservicio es desplegable de forma independiente.
- Persistencia: el monolito utiliza una única base de datos compartida. La propuesta adopta el patrón database-per-service, asignando una base de datos PostgreSQL propia e independiente a cada microservicio (ver sección 2.9).
- Comunicación: el monolito emplea invocaciones directas entre módulos. La propuesta utiliza comunicación asíncrona basada en eventos, donde el evento reserva_confirmada actúa como punto de desacoplamiento entre el procesamiento transaccional y los servicios consumidores.
- Escalabilidad: en el monolito, la escalabilidad es global y obliga a replicar toda la aplicación. En la propuesta, cada microservicio puede escalarse de forma horizontal en función de la demanda específica de su responsabilidad (ver RNF05).

- Resiliencia: en el monolito, una falla en cualquier componente impacta toda la aplicación. En la propuesta, los fallos quedan acotados al microservicio afectado, y los mecanismos de reintentos y dead-letter queue (ver RNF06) permiten manejar errores transitorios sin perder eventos.

Esta propuesta arquitectónica recoge los requerimientos funcionales y no funcionales definidos en las secciones 2.4 y 2.7, el modelo de datos especificado en la sección 2.9, y se concreta operativamente en el recurso computacional descrito en la siguiente sección.

2.3 Flujo principal del sistema

El flujo principal del sistema se inicia cuando el pasajero consulta vuelos disponibles, selecciona un itinerario, ingresa sus antecedentes personales y confirma un medio de pago simulado. A partir de esta información, el módulo central del sistema valida los datos, verifica disponibilidad, registra la reserva y genera la confirmación correspondiente.

Una vez completada exitosamente la reserva, el sistema publica —es decir, emite un mensaje asíncrono dirigido a todos los suscriptores interesados— el evento reserva_confirmada en Pub/Sub. Este evento constituye el punto de desacoplamiento entre el procesamiento transaccional principal y los servicios especializados que reaccionan posteriormente. A partir de él, el microservicio de Notificaciones gestiona el envío de la confirmación al pasajero, mientras que el microservicio de Recomendaciones procesa el mismo contexto para producir una sugerencia personalizada de upgrade de cabina o de servicios complementarios al viaje. Con ello, el flujo ilustra la transición desde una lógica centralizada hacia una coordinación distribuida basada en eventos (Kleppmann, 2017).

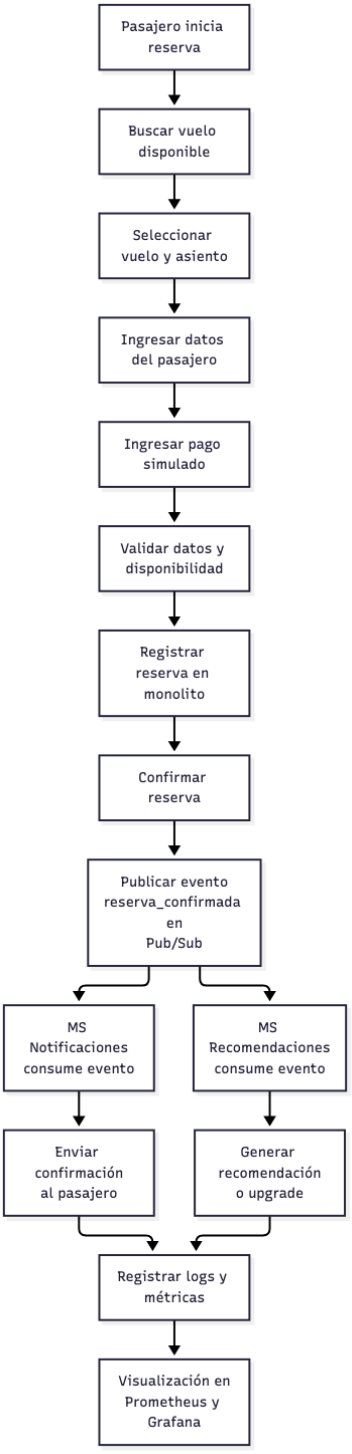


Figura 2-1. Flujo general del proceso de reserva y propagación de eventos

Fuente: elaboración propia.

La Figura 2-1 presenta el flujo principal del sistema, desde la solicitud de reserva hasta la publicación del evento reserva_confirmada y su posterior procesamiento por parte de los microservicios consumidores.

Cuando una recomendación de upgrade es aceptada por el pasajero, no se genera una nueva reserva, sino que la existente se actualiza: el microservicio de Reservas recibe la confirmación del cambio, modifica el estado correspondiente (por ejemplo, cabina y monto) y publica un nuevo evento reserva_actualizada, al que los demás servicios pueden suscribirse para mantener la consistencia.

2.4 Requerimientos funcionales

Los requerimientos funcionales definidos para la solución se estructuran en torno a actor, acción y objeto, según el formato sugerido para el trabajo. En este contexto, se establecen los siguientes requerimientos funcionales:

- RF01. Simular reserva de vuelo: El sistema le permitirá al pasajero realizar una reserva de vuelo ingresando los datos de itinerario, pasajero, asiento y pago dentro del módulo monolítico.
- RF02. Confirmar reserva y publicar evento: El sistema permitirá al módulo monolítico confirmar la reserva y publicar un evento en Pub/Sub cuando esta sea exitosa.
- RF03. Enviar notificación de confirmación: El microservicio de Notificaciones se suscribirá a los eventos de Pub/Sub y enviará una confirmación al pasajero por correo electrónico o por un canal simulado.
- RF04. Generar recomendaciones de upgrade: El microservicio de Recomendaciones procesa los eventos de reserva confirmada, aplica un modelo de propensión y generará sugerencias de upgrades o promociones para el pasajero.
Nota: El modelo de propensión es un clasificador estadístico que estima la probabilidad de que un pasajero acepte un upgrade o promoción a partir de su perfil, historial y contexto del viaje.
- RF05. Mantener independencia de datos: Cada microservicio deberá operar con su propia base de datos PostgreSQL, garantizando independencia en el almacenamiento y consistencia

eventual mediante la publicación y consumo de eventos en Pub/Sub, verificada en escenarios donde el microservicio consumidor refleja el cambio dentro de una ventana acotada tras la confirmación de la reserva.

Nota: La consistencia eventual entre microservicios se sustenta en tres elementos: (i) el evento reserva_confirmada se publica de manera atómica junto con la transacción de reserva, lo que evita la pérdida de eventos; (ii) Pub/Sub garantiza la entrega al menos una vez a los consumidores suscritos; y (iii) los consumidores implementan idempotencia (RNF07) para tolerar reentregas. En conjunto, estos mecanismos aseguran que el estado entre microservicios converja dentro de una ventana medible tras cada operación.

- RF06. Desplegar servicios en entorno local: El administrador del sistema podrá desplegar todos los servicios en un entorno cloud-native local mediante contenedores Docker orquestados en Kubernetes.
- RF07. Visualizar métricas de operación: El administrador del sistema podrá consultar métricas de rendimiento y disponibilidad a través de paneles de Prometheus y Grafana, permitiendo el monitoreo del sistema en tiempo real.

2.5 Actores y casos de uso

El actor principal del sistema es el pasajero, quien interactúa con la solución para consultar vuelos, registrar una reserva, recibir confirmación y acceder a recomendaciones. Como actor secundario se considera al administrador del sistema, responsable de desplegar la solución, supervisar su comportamiento y consultar métricas operacionales.

Los casos de uso principales corresponden a consultar vuelos disponibles, registrar reserva, confirmar reserva, publicar el evento de reserva confirmada, enviar notificación al pasajero, generar recomendación personalizada, desplegar servicios y monitorear métricas del sistema. En conjunto, estos casos de uso permiten representar tanto la dimensión funcional del negocio como la coordinación entre componentes internos.

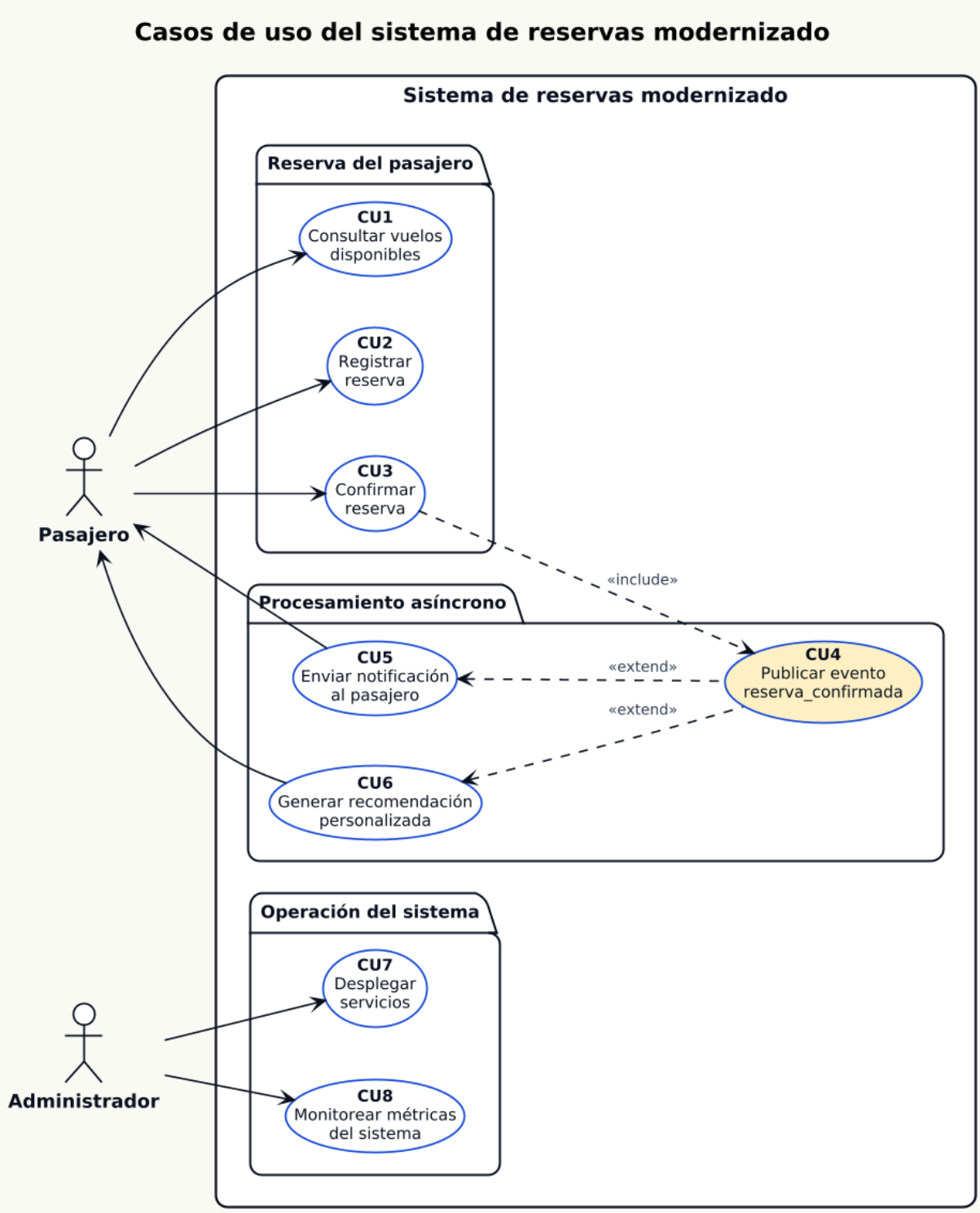


Figura 2-2. Casos de uso principales del sistema de reservas modernizado

Fuente: elaboración propia.

La Figura 2-2 resume las interacciones principales entre los actores del sistema y las funcionalidades esenciales de la solución propuesta.

2.6 Diagrama de secuencia

El diagrama de secuencia ilustra la interacción temporal entre el monolito, el bus de mensajería y los microservicios suscriptores durante el procesamiento del evento reserva_confirmada. Permite observar el orden de los mensajes asíncronos y la independencia operativa entre los componentes.

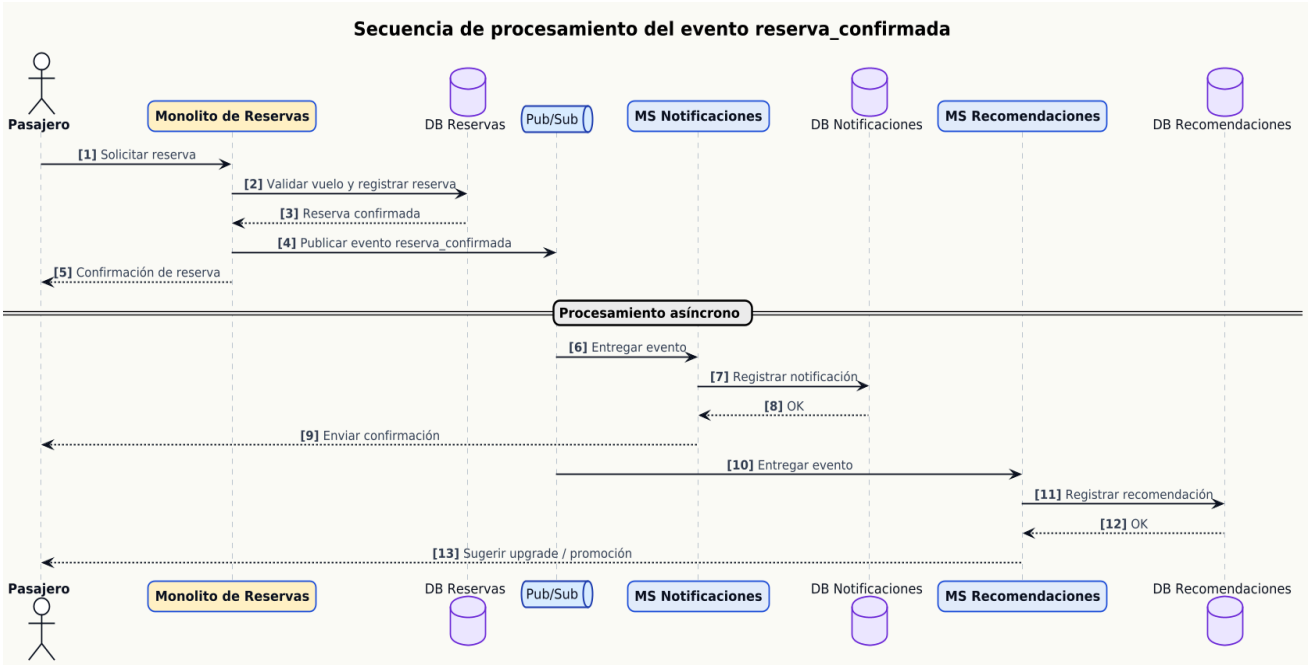


Figura 2-3. Secuencia de procesamiento del evento reserva_confirmada

Fuente: elaboración propia.

La Figura 2-3 describe la interacción temporal entre el monolito, el bus de mensajería y los microservicios a partir del evento reserva_confirmada.

2.7 Requerimientos no funcionales

La solución incorpora también requerimientos no funcionales orientados a asegurar una evaluación técnicamente consistente. En este sentido, se establecen los siguientes atributos de calidad, junto con su criterio de aceptación y forma de verificación:

- RNF01. Tiempo de respuesta: Para operaciones sincrónicas del monolito, como buscar vuelos y crear reservas, el sistema deberá alcanzar p95 <= 2.5 s y p99 <= 5 s bajo 50 RPS en entorno local. Verificación: prueba de carga con k6 o JMeter y métricas HTTP.
- RNF02. Latencia de eventos: Desde la publicación de reserva_confirmada hasta el procesamiento completo en el microservicio de Notificaciones, el sistema deberá mantener p95 <= 1 s con emulador Pub/Sub. Verificación: trazas con timestamp y logs correlacionados.
- RNF03. Throughput: El sistema deberá soportar al menos 100 eventos por minuto de forma sostenida, sin pérdida ni reintentos ilimitados. Verificación: contadores de consumo y DLQ (Dead Letter Queue, cola de mensajes fallidos) = 0.
- RNF04. Disponibilidad: Durante ventanas de prueba de dos horas, el sistema deberá mantener al menos 99 por ciento de endpoints críticos disponibles. Verificación: probes de liveness/readiness y uptime en Grafana.
- RNF05. Escalabilidad horizontal: Los microservicios deberán poder escalar horizontalmente, pasando de una instancia activa a tres instancias en paralelo (réplicas del mismo servicio) manteniendo p95 <= 2 s en endpoints y sin caídas. Verificación: escalado manual y pruebas repetidas.

Nota: Una réplica es una instancia adicional del mismo microservicio que se ejecuta en paralelo, compartiendo la carga de trabajo. En Kubernetes esto se controla mediante el campo réplicas del Deployment. El rango de una a tres se eligió por dos razones: (i) la línea base monolítica corre con una sola instancia, por lo que partir desde una réplica permite una comparación equivalente; y (ii) tres réplicas representan un escenario de carga moderada-alta sostenible dentro de los recursos disponibles en el entorno local de pruebas, sin saturar el cluster de Kubernetes ni introducir variabilidad por contención de recursos en el host.

- RNF06. Resiliencia y reintentos: Ante un fallo transitorio del adaptador de correo, el sistema deberá aplicar reintentos exponenciales de máximo tres intentos y, si persiste la falla, enviar el mensaje a DLQ sin duplicar notificaciones. Verificación: pruebas de caos y revisión de DLQ.
- RNF07. Idempotencia: Los mensajes duplicados con el mismo eventId no deberán producir efectos repetidos. Verificación: prueba con mensajes duplicados y aserción en repositorio.
- RNF08. Seguridad básica: La comunicación interna se realizará por HTTP local, ocultando datos sensibles en logs y evitando persistir credenciales en texto plano. Verificación: revisión de código y escaneo estático.
- RNF09. Observabilidad: Todos los servicios deberán exponer /metrics para Prometheus y disponer de tableros en Grafana con latencia, throughput, errores y colas. Verificación: chequeo de endpoints y panel cargado.
- RNF10. Registro y trazabilidad: El sistema deberá correlacionar operaciones mediante X-Request-Id y/o eventId, manteniendo logs al menos por siete días en entorno local. Verificación: configuración de logs y consulta de trazas.
- RNF11. Mantenibilidad: El build completo deberá finalizar en cinco minutos o menos y las pruebas en ocho minutos o menos en CI local. Verificación: pipeline en Jenkins o GitHub Actions.
- RNF12. Portabilidad: El sistema deberá ejecutarse en Minikube y K3s sin cambios de código, modificando solo manifiestos o configuración de despliegue. Verificación: despliegues en ambos entornos.
- RNF13. Integridad de datos: Cada servicio deberá mantener consistencia ACID en su propia base de datos, mientras que la consistencia entre servicios será eventual en un máximo de cinco segundos tras la publicación del evento. Verificación: pruebas de consistencia post-evento.
- RNF14. Capacidad: Las bases de datos deberán dimensionarse para al menos 100 mil reservas simuladas y 500 mil eventos sin degradación crítica del rendimiento. Verificación: carga sintética y monitoreo de I/O.
- RNF15. Usabilidad mínima: Las respuestas de error deberán incluir código y mensaje claro, y la documentación de endpoints deberá mantenerse en OpenAPI. Verificación: revisión de API y pruebas de contrato.

- RNF16. Confiabilidad de entrega: La tasa de error en la entrega de notificaciones deberá ser menor a 1 por ciento, con reintentos automáticos hasta tres veces antes de enviar a DLQ. Verificación: KPI en tablero y auditoría de entregas.

2.8 Información que manejará el sistema

Las entradas del sistema corresponden al formulario de reserva completado por el pasajero, a los mensajes asincrónicos publicados en Pub/Sub y a los archivos y parámetros de configuración requeridos para el despliegue local y el monitoreo. Asimismo, pueden considerarse como entradas complementarias determinados parámetros de evaluación técnica, tales como perfiles de carga, umbrales de rendimiento o configuraciones de observabilidad.

Las salidas del sistema incluyen la confirmación de la reserva, las recomendaciones personalizadas generadas por el microservicio correspondiente, los logs de operación, las trazas de correlación, las métricas expuestas a Prometheus y visualizadas en Grafana, así como los reportes comparativos que apoyan el análisis académico del trabajo.

2.9 Entidades de datos y persistencia

Desde la perspectiva del modelo de datos, las entidades principales persistidas son Pasajero, Vuelo, Reserva, Asiento, Pago, Notificación y Recomendación. Estas entidades permiten representar el flujo transaccional principal y los procesos complementarios. De manera complementaria, EventoReserva corresponde al mensaje asíncrono publicado en el bus Pub/Sub tras la confirmación de una reserva; no se materializa como tabla persistida, sino que viaja por el bus y es consumido por los microservicios de Notificaciones y Recomendaciones, los cuales registran sus propios resultados en Notificación y Recomendación, respectivamente.

Se propone un modelo de persistencia relacional soportado por PostgreSQL, adecuado para mantener consistencia en operaciones de reserva y trazabilidad de estados. En la arquitectura modernizada cada servicio mantiene una base de datos PostgreSQL propia e independiente. Esto significa que cada microservicio dispone de una instancia distinta de

PostgreSQL, con su propio motor en ejecución, su propio esquema de tablas y su propio ciclo de vida, y que no se trata de réplicas de una misma base de datos. Una réplica corresponde a una copia sincronizada de la base original que contiene exactamente los mismos datos, y suele utilizarse para alta disponibilidad o para distribuir consultas de lectura. Una instancia separada, en cambio, es una base de datos completamente distinta, con datos propios y aislados, que solo se comunica con las demás mediante la publicación y consumo de eventos. Este diseño busca evitar el acoplamiento directo a nivel de persistencia y favorecer la autonomía funcional de los componentes (Kleppmann, 2017).

La decisión de adoptar una base de datos propia por microservicio responde a tres objetivos específicos de este proyecto. Primero, garantizar la autonomía operativa de cada servicio, de modo que la evolución del esquema de Reservas no obligue a coordinar despliegues con Notificaciones o Recomendaciones. Segundo, aislar fallos: una falla en la base de un servicio no compromete la disponibilidad de los demás, lo que es coherente con los requisitos no funcionales de disponibilidad definidos previamente. Tercero, validar empíricamente, dentro del entorno local de Kubernetes, el patrón database-per-service y el rol de los eventos como mecanismo de integración, demostrando que la consistencia eventual entre servicios resulta viable para el dominio de reservas aéreas.

Este patrón de persistencia distribuida corresponde al enfoque database-per-service, uno de los pilares del estilo arquitectónico de microservicios (Newman, 2021), y se adopta aquí por su alineamiento directo con los objetivos específicos del proyecto. Su adopción implica, deliberadamente, renunciar a las garantías de integridad referencial física entre servicios: las relaciones Reserva↔Notificación y Reserva↔Recomendación se preservan como claves lógicas transportadas en los eventos, no como claves foráneas en la base de datos. En consecuencia, las consultas transversales entre dominios dejan de resolverse mediante JOINs cruzados y pasan a resolverse por composición en el borde (API composition o BFF) o por vistas materializadas construidas desde el bus de eventos (patrón CQRS), siendo esta última una línea de trabajo futuro; para el alcance de este trabajo, las necesidades de agregación quedan cubiertas por la observabilidad centralizada en Prometheus y Grafana, descrita en la sección 2.11 y materializada en la vista de monitoreo del administrador (Figura 2-10).

Asimismo, el patrón admite persistencia poliglota — motores distintos por servicio según su carga —, pero en este alcance se opta por un uso homogéneo de PostgreSQL en los tres

microservicios con el fin de reducir la carga operacional del entorno local de desarrollo y porque las cargas de trabajo son de naturaleza OLTP similar; la adopción de motores especializados (por ejemplo, un almacén clave-valor para notificaciones de alta frecuencia o un motor columnar para analítica de recomendaciones) se identifica como una posible evolución futura. Esta síntesis acompaña la lectura del modelo lógico integrado que se presenta a continuación en la Figura 2-4.

La Figura 2-4 representa, en forma de modelo lógico integrado, las entidades principales del dominio y sus relaciones más relevantes. Si bien el diagrama incluye tipos de datos tentativos a modo ilustrativo, no corresponde a un modelo físico desplegado: en la arquitectura real, cada microservicio mantiene su propia base de datos independiente, por lo que las relaciones entre entidades de distintos microservicios (por ejemplo, entre Reserva y Recomendación o Notificación) no se materializan como claves foráneas físicas, sino que se preservan de manera lógica a través de los identificadores transportados por los eventos del bus Pub/Sub. La figura sirve, por tanto, como apoyo conceptual para describir la persistencia del sistema, no como esquema físico desplegado.

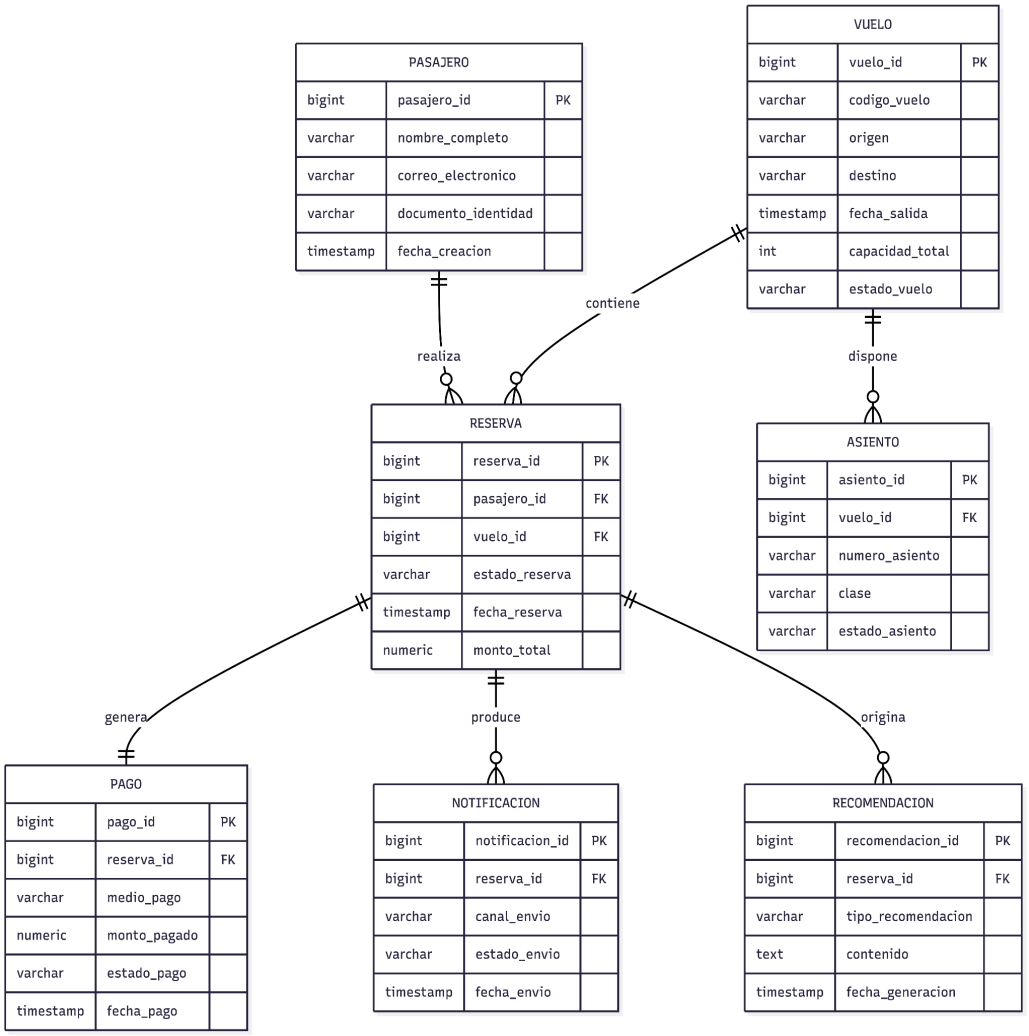


Figura 2-4. Modelo de datos relacional del sistema de reservas

Fuente: elaboración propia.

Para efectos del alcance de este trabajo, el modelo de datos presentado es una versión simplificada que recoge solo los campos esenciales para ilustrar el flujo del sistema. Se omiten atributos como la nacionalidad del pasajero, la diferenciación entre clases de cabina, o representaciones específicas de aeropuertos/ciudades en origen y destino, los cuales en una implementación productiva deberían integrarse a partir de entidades especializadas. De igual forma, el campo fecha_salida incluye tanto fecha como hora, y los identificadores asiento_id y numero_asiento corresponden, respectivamente, al identificador interno único del asiento en la base de datos y a su numeración visible para el pasajero. Asimismo, se asume una cardinalidad uno a uno entre reserva y asiento: cada reserva corresponde a un único asiento

para un único pasajero, por lo que no se requiere una tabla intermedia pasajero–reserva–asiento; en un escenario de reservas grupales esta relación pasaría a ser uno a muchos y demandaría una entidad asociativa específica. Finalmente, cabe recordar que, al tener cada microservicio su propia base de datos, las relaciones aparentes entre entidades de distintos microservicios (por ejemplo, Reserva ↔ Notificación o Reserva ↔ Recomendación) no se implementan como claves foráneas físicas, sino que se mantienen de manera lógica mediante los identificadores transportados por los eventos del bus Pub/Sub.

Operativamente, esta integración lógica entre microservicios se implementa como sigue. Cuando el microservicio de Reservas confirma una reserva, (i) ejecuta la transacción local que persiste la fila en su propia tabla reserva y, de manera atómica, (ii) publica en Pub/Sub el evento reserva_confirmada, cuyo payload incluye el reserva_id generado por la base de datos de Reservas junto con los campos del dominio necesarios para los consumidores (pasajero_id, vuelo_id, asiento asignado, monto). El microservicio de Notificaciones, suscrito a ese tipo de evento, lo consume, extrae el reserva_id del payload y persiste su propio registro en la tabla notificacion incluyendo ese reserva_id como columna propia; el microservicio de Recomendaciones procede de manera análoga sobre su tabla recomendación. Dicho reserva_id no es una clave foránea desde el punto de vista del motor de base de datos — ninguna restricción referencial lo valida contra la tabla reserva de Reservas, que vive en otra instancia PostgreSQL —, sino un identificador opaco que preserva la trazabilidad lógica. En consecuencia, una consulta como "todas las notificaciones de una reserva" se resuelve localmente en la base de Notificaciones mediante un SELECT ... WHERE reserva_id = X, sin requerir acceso a la base de Reservas. Los cambios posteriores en una reserva (por ejemplo, cancelación o actualización por upgrade) se propagan por el mismo mecanismo, publicando eventos adicionales (reserva_cancelada, reserva_actualizada) que los consumidores aplican sobre sus propios registros respetando la misma política de idempotencia.

2.9.1 Tablas principales del sistema

Con el propósito de reforzar la descripción del modelo de datos, se consideran como tablas principales pasajero, vuelo, reserva, asiento, pago, notificación y recomendación. La tabla pasajero almacena la información de identificación y contacto del usuario; la tabla vuelo representa la oferta disponible; reserva concentra la operación principal del negocio y se relaciona con pasajero y vuelo; pago registra el medio y estado del pago simulado; mientras que notificación y recomendación conservan los resultados de los procesos asincrónicos posteriores a la confirmación de la reserva.

Esta descripción no busca agotar el detalle físico de la base de datos, sino explicitar las estructuras esenciales que permiten comprender la persistencia del sistema dentro del alcance técnico del trabajo.

2.9.1.1 Tabla principal: reserva

- Nombre lógico: Reserva
- Nombre físico: reserva
- Descripción: almacena la operación principal de reserva realizada por un pasajero sobre un vuelo.
- Clave primaria: reserva_id
- Claves foráneas: pasajero_id (hacia tabla pasajero), vuelo_id (hacia tabla vuelo)

Nombre del campo	**Tipo**	**Descripción**
reserva_id	BIGINT	Identificador correlativo autogenerado por la base de datos, de tipo entero incremental, asignado al momento de registrar la reserva.
pasajero_id	BIGINT	Referencia al pasajero que realiza la reserva (FK hacia tabla pasajero).
vuelo_id	BIGINT	Referencia al vuelo seleccionado (FK hacia tabla vuelo).
estado_reserva	CHAR(1)	Estado actual de la reserva codificado en un carácter: 1=confirmada, 2=pendiente, 3=cancelada, 4=expirada.
fecha_reserva	TIMESTAMP	Fecha y hora de creación de la reserva.
monto_total	DECIMAL(12,2)	Valor asociado al pago simulado.

2.9.1.2 Tabla principal: vuelo

- Nombre lógico: Vuelo
- Nombre físico: vuelo
- Descripción: representa la oferta de vuelo disponible dentro del sistema.
- Clave primaria: vuelo_id

Nombre del campo	**Tipo**	**Descripción**
vuelo_id	BIGINT	Identificador interno correlativo autogenerado por la base de datos, de tipo entero incremental, distinto del código comercial del vuelo (campo codigo_vuelo).
codigo_vuelo	VARCHAR(10)	Código comercial del vuelo.
origen	VARCHAR(100)	Ciudad o aeropuerto de origen.
destino	VARCHAR(100)	Ciudad o aeropuerto de destino.
fecha_salida	TIMESTAMP	Fecha y hora programada de salida.
capacidad_total	INT	Número total de asientos.
estado_vuelo	CHAR(1)	Estado operativo del vuelo codificado en un carácter: 1=programado, 2=embarcando, 3=en vuelo, 4=aterrizado, 5=cancelado, 6=retrasado.

2.9.1.3 Tabla principal: pasajero

- Nombre lógico: Pasajero
- Nombre físico: pasajero
- Descripción: almacena la información de identificación y contacto del pasajero.
- Clave primaria: pasajero_id

Nombre del campo	**Tipo**	**Descripción**
pasajero_id	BIGINT	Identificador único del pasajero.
nombre	VARCHAR(80)	Nombre del pasajero.
apellido	VARCHAR(80)	Apellido del pasajero.
correo	VARCHAR(120)	Correo electrónico de contacto.
telefono	VARCHAR(20)	Número de teléfono de contacto.
documento	VARCHAR(20)	Documento de identidad (por ejemplo, RUT o pasaporte).

2.9.1.4 Tabla principal: asiento

- Nombre lógico: Asiento
- Nombre físico: asiento
- Descripción: modela la disponibilidad y asignación de cada ubicación del vuelo.
- Clave primaria: asiento_id
- Claves foráneas: vuelo_id (hacia tabla vuelo)

Nombre del campo	**Tipo**	**Descripción**
asiento_id	BIGINT	Identificador único interno del asiento.
vuelo_id	BIGINT	Referencia al vuelo al que pertenece el asiento (FK hacia tabla vuelo).
numero_asiento	VARCHAR(5)	Numeración visible para el pasajero (por ejemplo, 12A).
clase	VARCHAR(10)	Clase del asiento (economy, business o first).
estado	VARCHAR(20)	Estado operativo del asiento (disponible, reservado, ocupado o bloqueado).

2.9.1.5 Tabla principal: pago

- Nombre lógico: Pago
- Nombre físico: pago
- Descripción: registra el medio y el estado del pago simulado asociado a cada reserva.
- Clave primaria: pago_id
- Claves foráneas: reserva_id (hacia tabla reserva)

Nombre del campo	**Tipo**	**Descripción**
pago_id	BIGINT	Identificador único del pago.
reserva_id	BIGINT	Referencia a la reserva asociada al pago (FK hacia tabla reserva).
monto	DECIMAL(10,2)	Monto cobrado por la reserva.
moneda	VARCHAR(3)	Código ISO 4217 de la moneda (por ejemplo, CLP, USD).
medio_pago	VARCHAR(20)	Medio de pago simulado (tarjeta, transferencia o voucher).
estado	VARCHAR(20)	Estado del pago (pendiente, confirmado, rechazado o reembolsado).
fecha_pago	DATETIME	Fecha y hora del registro del pago.

2.9.1.6 Tabla principal: notificación

- Nombre lógico: Notificación
- Nombre físico: notificacion
- Descripción: conserva el resultado de los mensajes emitidos al pasajero por el microservicio de Notificaciones.
- Clave primaria: notificacion_id
- Clave lógica: reserva_id (identificador transportado por el evento, sin clave foránea física entre bases de datos de microservicios).

Nombre del campo	**Tipo**	**Descripción**
notificacion_id	BIGINT	Identificador único de la notificación.
reserva_id	BIGINT	Identificador de la reserva asociada, transportado por el evento (relación lógica entre microservicios).
canal	VARCHAR(20)	Canal de envío (correo, SMS o push).
destinatario	VARCHAR(120)	Dirección o número de destino del mensaje.
contenido	TEXT	Contenido textual del mensaje enviado.
estado	VARCHAR(20)	Estado del envío (pendiente, enviado o fallido).
fecha_envio	DATETIME	Fecha y hora en que se emitió la notificación.

2.9.1.7 Tabla principal: recomendación

- Nombre lógico: Recomendación
- Nombre físico: recomendacion
- Descripción: conserva el resultado de las sugerencias generadas por el microservicio de Recomendaciones.
- Clave primaria: recomendacion_id
- Clave lógica: reserva_id (identificador transportado por el evento, sin clave foránea física entre bases de datos de microservicios).

Nombre del campo	**Tipo**	**Descripción**
recomendacion_id	BIGINT	Identificador único de la recomendación.
reserva_id	BIGINT	Identificador de la reserva asociada, transportado por el evento (relación lógica entre microservicios).
tipo	VARCHAR(20)	Tipo de recomendación (upgrade o promoción).
criterio	VARCHAR(80)	Criterio o regla aplicada por el modelo (por ejemplo, perfil del pasajero o historial).
detalle	TEXT	Descripción textual de la sugerencia generada.
estado	VARCHAR(20)	Estado de la propuesta (generada, aceptada, rechazada o expirada).
fecha_generacion	DATETIME	Fecha y hora de generación de la recomendación.

2.10 Metodología, alcance y resultados esperados

La metodología del proyecto se plantea bajo un enfoque iterativo e incremental, permitiendo avanzar desde la comprensión del sistema monolítico hacia el diseño de la arquitectura modernizada y su posterior evaluación técnica. El alcance del trabajo se circunscribe a la simulación y modernización de un sistema de reservas de vuelos en un entorno controlado, sin integración con aerolíneas reales, pasarelas de pago productivas ni sistemas externos de operación comercial.

Como resultado del trabajo se espera obtener una caracterización clara del sistema base, una arquitectura de microservicios debidamente documentada, diagramas de arquitectura y secuencia, una propuesta de persistencia distribuida y evidencia técnica que permita observar beneficios tales como mayor desacoplamiento funcional, independencia de despliegue y mejor trazabilidad operativa.

2.11 Recurso computacional, software y vistas del sistema

El sistema se desarrolla y despliega en un entorno local controlado, diseñado para reproducir, en escala reducida, las condiciones de una arquitectura cloud-native. El desarrollo se apoya en Java 17 (monolito) a 21 (microservicios), Spring Boot y principios de arquitectura hexagonal. La comunicación asincrónica se implementa mediante Google Cloud Pub/Sub con emulador local, mientras que la persistencia utiliza PostgreSQL.

Para el empaquetado y despliegue se emplean Docker y Kubernetes, a través de Minikube o K3s. La observabilidad se aborda con Prometheus y Grafana, herramientas que permiten recolectar métricas, visualizar indicadores y apoyar la evaluación del sistema en los distintos escenarios de prueba (Burns et al., 2022; Kim et al., 2021).

2.11.1 Configuración general del recurso computacional

El recurso computacional considerado corresponde a un equipo de desarrollo personal con capacidad suficiente para ejecutar contenedores, bases de datos, un emulador de mensajería y un cluster local de Kubernetes. De manera referencial, se requiere un procesador multinúcleo, memoria RAM adecuada para la ejecución concurrente de varios servicios y almacenamiento SSD que favorezca tiempos de respuesta estables durante compilación, pruebas y despliegues.

2.11.2 Diagrama de menús

Dado que el proyecto se concentra principalmente en el análisis y el diseño del sistema, el diagrama de menús se plantea en forma simple y jerárquica, describiendo las opciones funcionales que el pasajero y el administrador pueden ejecutar sobre la solución simulada, de manera coherente con las vistas de usuario presentadas en la Figura 2-6 y siguientes.La estructura considerada parte de un menú principal con las opciones: Consulta de vuelos (que inicia el flujo transaccional), Registro de reserva (que agrupa internamente los pasos de ingreso de datos del pasajero, selección de asiento, pago simulado y confirmación), Recomendaciones (que presenta las sugerencias generadas por el microservicio correspondiente) y Monitoreo del sistema (reservado a perfiles con rol de administrador, con acceso a métricas y trazas). La pantalla de inicio de sesión se considera un paso previo de autenticación y no se incorpora como opción del menú principal.

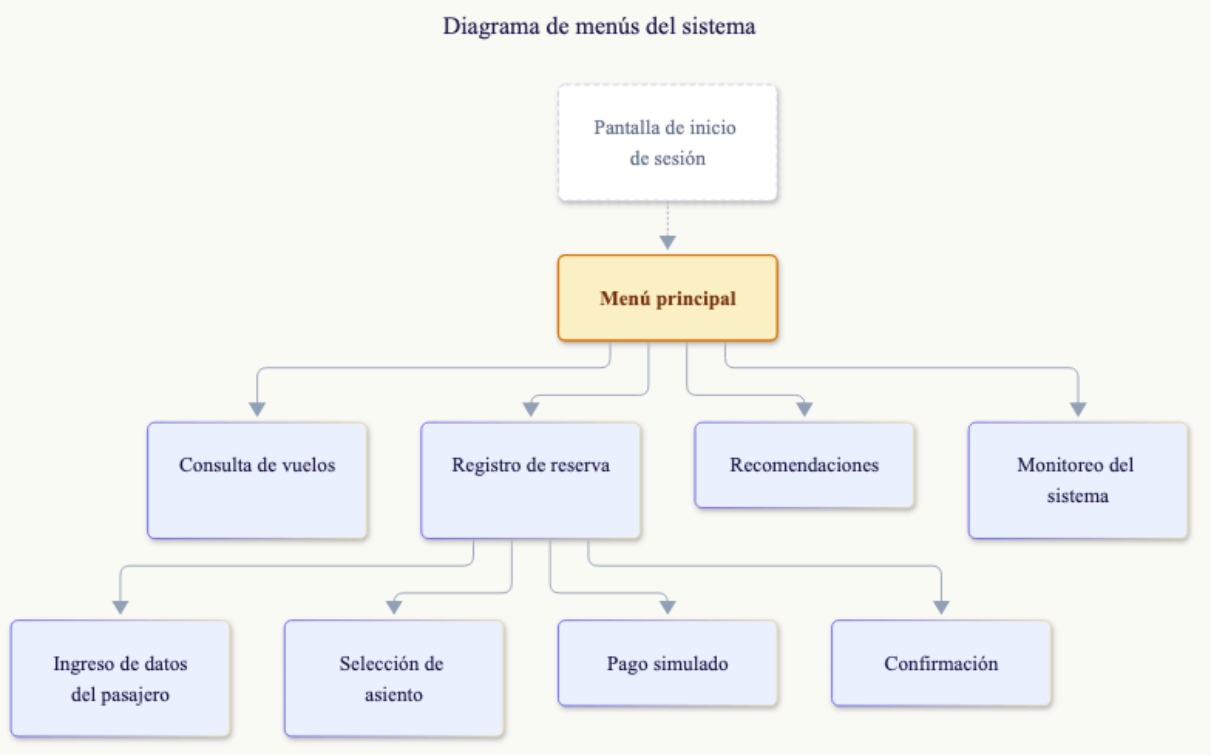


Figura 2-5. Diagrama de menús del sistema

Fuente: elaboración propia.

La Figura 2-5 organiza las opciones funcionales principales del sistema y muestra la estructura general de navegación considerada para la solución propuesta.

2.11.3 Vistas de usuario

Las vistas de usuario consideradas con fines ilustrativos corresponden a la pantalla de consulta de vuelos, la pantalla de registro de reserva, la vista de confirmación de la reserva, la vista de recomendaciones generadas y el panel de monitoreo de métricas. Estas vistas no persiguen constituir una interfaz final completa, sino aportar evidencia visual que facilite la comprensión del funcionamiento del sistema propuesto.

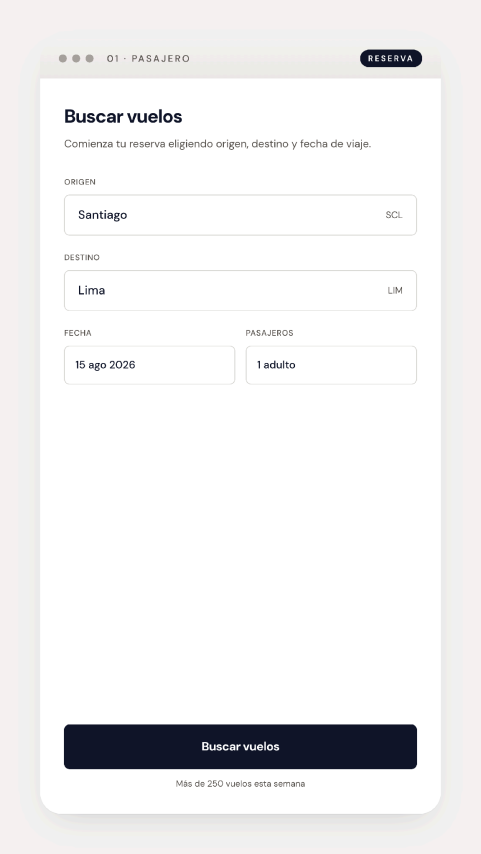


Figura 2-6. Pantalla de búsqueda de vuelos

Fuente: elaboración propia.

La Figura 2-6 ilustra la pantalla inicial del flujo de reserva, en la que el pasajero ingresa origen, destino, fecha y número de pasajeros para generar la consulta de vuelos disponibles. Esta vista corresponde al caso de uso CU1: Consultar vuelos disponibles.

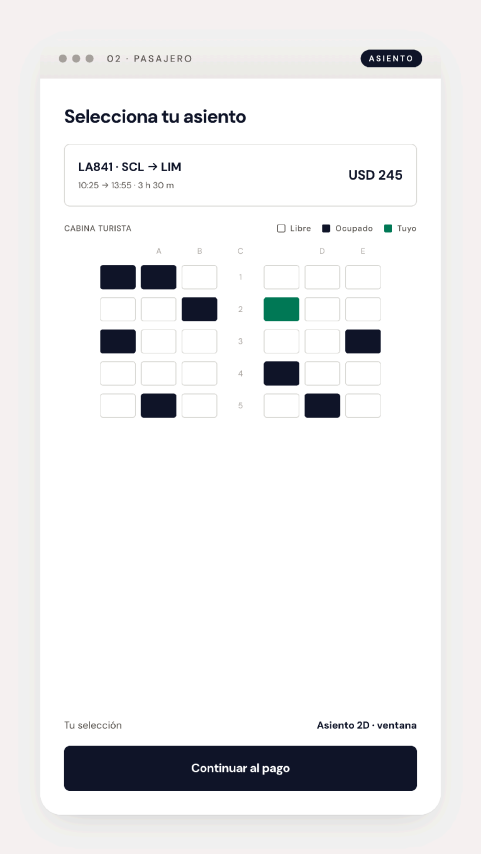


Figura 2-7. Pantalla de selección de asiento

Fuente: elaboración propia.

La Figura 2-7 muestra la pantalla de selección de asiento sobre el vuelo elegido. El mapa de cabina diferencia visualmente entre asientos disponibles, ocupados y la selección actual del pasajero. Esta vista corresponde al caso de uso CU2: Reservar asiento.

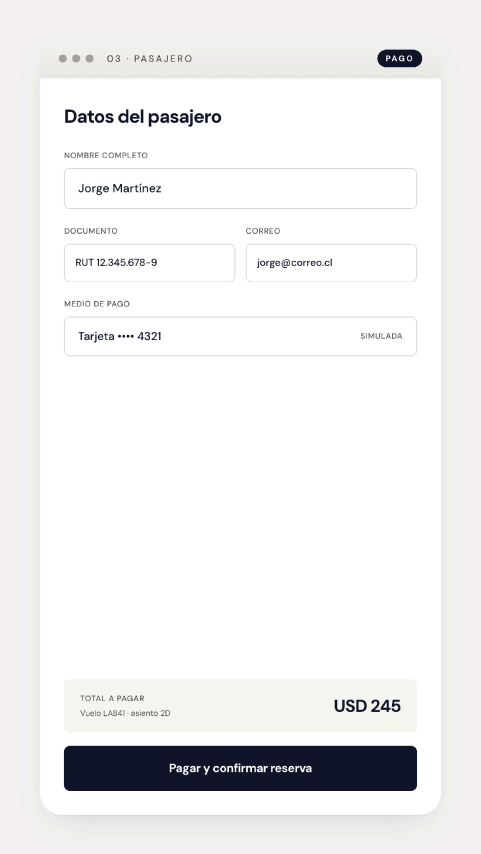


Figura 2-8. Pantalla de datos del pasajero y pago

Fuente: elaboración propia.

La Figura 2-8 presenta el formulario donde el pasajero confirma sus datos personales, ingresa su medio de pago simulado y revisa el monto total antes de finalizar la operación. Esta vista corresponde al caso de uso CU3: Confirmar pago simulado.

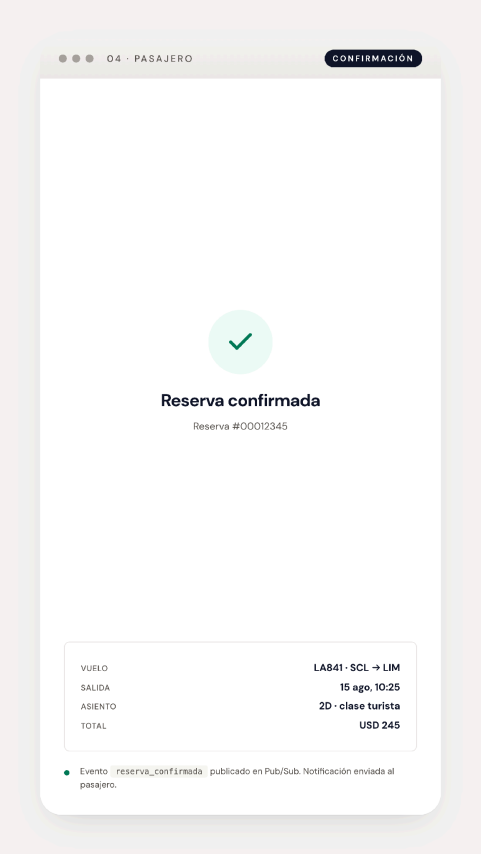


Figura 2-9. Pantalla de confirmación de reserva

Fuente: elaboración propia.

La Figura 2-9 muestra la pantalla de confirmación que recibe el pasajero una vez procesado el pago. Incluye el detalle final de la reserva y notifica la publicación del evento reserva_confirmada en Pub/Sub. Esta vista representa el cierre del flujo del pasajero y se relaciona con los casos de uso CU3 y CU4.

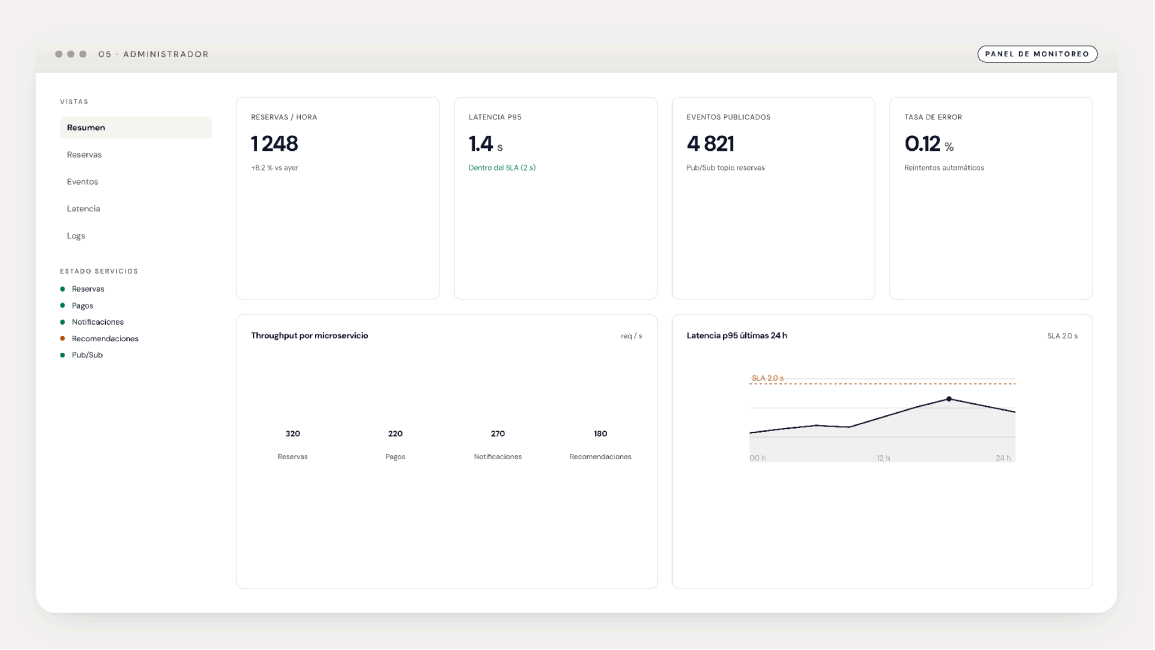


Figura 2-10. Panel de monitoreo del administrador

Fuente: elaboración propia.

La Figura 2-10 corresponde al panel de monitoreo destinado al administrador del sistema. Reúne indicadores agregados como reservas por hora, latencia p95, volumen de eventos publicados, tasa de error y estado operativo por microservicio. Esta vista corresponde al caso de uso CU7: Monitorear métricas del sistema.

2.12 Síntesis del capítulo

Los aspectos relevantes del diseño desarrollados en este capítulo permiten traducir la estrategia de modernización a componentes concretos de datos, integración, infraestructura y representación funcional. Sobre esta base, el trabajo queda en condiciones de avanzar hacia la evaluación comparativa entre el enfoque monolítico y la arquitectura modernizada.

Capítulo 3. Evaluación técnica y resultados

3.1 Objetivo de la evaluación

La evaluación técnica tiene por finalidad analizar, en un entorno controlado, las diferencias observables entre el sistema monolítico de reservas de vuelos y la arquitectura modernizada basada en microservicios con comunicación asincrónica. En particular, se busca determinar en qué medida la solución propuesta aporta mejoras en desacoplamiento funcional, facilidad de despliegue, trazabilidad operativa, resiliencia y capacidad de escalamiento.

3.2 Escenarios de prueba y métricas

Para desarrollar esta comparación se consideran, al menos, cinco escenarios de prueba: reserva individual, propagación de eventos, carga sostenida, falla controlada y despliegue-operación del sistema. Asociadas a estos escenarios se encuentran métricas tales como tiempo de respuesta, latencia de eventos, throughput, disponibilidad, resiliencia, observabilidad y facilidad de despliegue.

La definición de estos escenarios permite mantener coherencia entre los requerimientos no funcionales establecidos y la evidencia que se espera observar al final del trabajo. En consecuencia, la evaluación no se presenta como una etapa aislada, sino como una extensión natural del análisis y del diseño desarrollados en los capítulos previos.

3.3 Análisis comparativo entre enfoques

Desde una perspectiva estructural, el sistema monolítico presenta la ventaja de centralizar el flujo de negocio en una única aplicación, lo que simplifica el despliegue inicial y la consistencia directa de las operaciones. Sin embargo, este mismo modelo concentra dependencias internas que limitan la independencia de despliegue, la escalabilidad selectiva y la separación de responsabilidades funcionales.

La arquitectura de microservicios, en cambio, incorpora una mayor complejidad operativa inicial, ya que requiere mensajería, observabilidad, despliegue distribuido y coordinación entre componentes. No obstante, esta complejidad se encuentra justificada cuando se persigue desacoplar procesos complementarios, aislar fallos parciales y facilitar la evolución progresiva del sistema (Newman, 2021; Ford et al., 2017). En consecuencia, la comparación entre ambos enfoques debe entenderse de manera contextual y no como una afirmación absoluta de superioridad arquitectónica.

3.4 Presentación e interpretación de resultados

Los resultados de la evaluación se presentan mediante tablas comparativas y párrafos interpretativos que distinguen entre la evidencia cualitativa ya observable en el diseño de la solución y las métricas cuantitativas que todavía requieren ejecución experimental controlada. Esta distinción permite evitar sobre interpretaciones y mantener coherencia entre los requerimientos no funcionales definidos y el nivel de validación efectivamente alcanzado.

Dentro de este marco, las tablas siguientes no buscan demostrar una superioridad absoluta de la arquitectura de microservicios, sino exponer de manera trazable qué aspectos muestran ventajas conceptuales o de diseño en el enfoque modernizado, qué costos operativos introduce y qué mediciones permanecen abiertas para una validación posterior. La evidencia se interpreta, por tanto, dentro de una simulación académica y controlada, con resultados parciales y delimitaciones explícitas (Hernández Sampieri et al., 2014).

Tabla 3-1. Matriz de evaluación comparativa

Atributo	Métrica	Monolito	Microservicios	Fuente de evidencia	Observación
Tiempo de respuesta	p95 reserva completa	Sin medición cuantitativa consolidada en esta etapa	Sin medición cuantitativa consolidada en esta etapa	Diseño de pruebas con k6 o JMeter	La comparación numérica quedó definida como validación futura.

Latencia de eventos	publish a consumo completo	No aplica, por procesamiento centralizado	Flujo asíncrono diseñado, sin trazas consolidadas de extremo a extremo	Diseño del flujo, logs y timestamps planificados	Métrica pertinente sólo para la arquitectura basada en eventos.
Throughput	reservas por minuto	Escenario de carga definido, sin medición consolidada	Escenario de carga definido, sin medición consolidada	Plan de pruebas de carga	Existe criterio de evaluación, pero no un valor final comparable.
Resiliencia	impacto ante falla parcial	Mayor impacto potencial ante fallas internas por concentración funcional	Mayor aislamiento esperado por desacoplamiento y mensajería asíncrona	Arquitectura propuesta y escenario de falla controlada	Conclusión cualitativa sustentada en el diseño; pendiente validación experimental.
Observabilidad	métricas y trazas	Observabilidad limitada y centralizada en una sola aplicación	Observabilidad más rica y distribuida con Prometheus y Grafana	Arquitectura propuesta, endpoints y tableros definidos	Existe evidencia de diseño e instrumentación prevista para la solución

					modernizada.
Despliegue	pasos o tiempo de despliegue	Despliegue unitario de menor complejidad inicial	Despliegue distribuido con mayor complejidad, pero con mayor independencia evolutiva	Configuración de despliegue local con Docker y Kubernetes	Comparación cualitativa basada en la configuración propuesta.

Fuente: elaboración propia.

La Tabla 3-1 muestra que, al momento del cierre del trabajo, la principal fortaleza de la evidencia disponible no reside en valores numéricos definitivos, sino en la trazabilidad entre escenarios de prueba, atributos de calidad y mecanismos de observación ya definidos. En particular, observabilidad, despliegue y resiliencia presentan diferencias cualitativas identificables, mientras que tiempo de respuesta y throughput requieren mediciones comparables para emitir juicios cuantitativos robustos.

Tabla 3-2. Interpretación cualitativa de los enfoques comparados

Dimensión	Monolito	Microservicios	Interpretación
Acoplamiento	Alto	Menor, basado en eventos	Favorece la evolución independiente de servicios
Complejidad operativa	Menor al inicio	Mayor al inicio	Requiere más herramientas y coordinación
Escalabilidad	Global	Selectiva por servicio	Permite ajustar recursos según la carga de cada servicio
Resiliencia	Más sensible a fallas internas	Mayor aislamiento esperado	Depende de la implementación concreta y de la estrategia de manejo de errores
Integración futura	Más rígida	Más flexible	Facilita nuevas integraciones y extensiones funcionales
Observabilidad	Concentrada	Distribuida y más rica	Mejora la trazabilidad operativa y el análisis de eventos

Fuente: elaboración propia.

La Tabla 3-2 complementa el análisis al evidenciar que el enfoque de microservicios ofrece mejores condiciones para el desacoplamiento, la evolución independiente de componentes y la trazabilidad operativa, pero a costa de una mayor complejidad de despliegue, monitoreo y coordinación entre servicios. Por ello, la conveniencia de la modernización depende no solo de sus beneficios técnicos esperados, sino también de la capacidad del equipo para asumir dichas exigencias.

Dado que la evidencia disponible en esta sección es principalmente comparativa y parcialmente cualitativa, no se incorporan gráficos cuantitativos adicionales. La inclusión de curvas, histogramas o paneles de resultados consolidados sería pertinente únicamente cuando existan mediciones homogéneas, repetibles y comparables para ambos enfoques.

3.5 Síntesis del capítulo

La evaluación técnica desarrollada en este capítulo permite sintetizar los principales hallazgos del trabajo. Primero, el diseño de escenarios de prueba y la definición de métricas (latencia, disponibilidad, tolerancia a fallos y acoplamiento) establecieron un marco común para comparar el enfoque monolítico y el de microservicios desde una perspectiva técnica reproducible. Segundo, el análisis comparativo evidenció que el enfoque de microservicios con comunicación asíncrona ofrece mejores condiciones para el desacoplamiento, la evolución independiente de componentes y la trazabilidad operativa, pero a costa de una mayor complejidad de despliegue, monitoreo y coordinación entre servicios. Tercero, la interpretación de los resultados, de carácter principalmente cualitativo y comparativo dentro del alcance de este trabajo, confirmó que la modernización aporta beneficios concretos en términos de escalabilidad y mantenibilidad, siempre que el equipo esté preparado para asumir las exigencias asociadas. En consecuencia, el capítulo cierra con una respuesta afirmativa pero condicionada a los objetivos específicos del proyecto: los microservicios constituyen una alternativa recomendable cuando el contexto organizacional, operativo y técnico lo justifica, y no deben adoptarse como una solución universal.

Conclusiones y observaciones

El trabajo desarrollado permite concluir que la modernización de un sistema monolítico de reservas de vuelos hacia una arquitectura de microservicios con comunicación asincrónica constituye una estrategia técnicamente pertinente cuando el objetivo es desacoplar responsabilidades complementarias, mejorar la trazabilidad operativa y preparar el sistema para una evolución progresiva. A partir del análisis del sistema base y del diseño de la solución propuesta, se identificó que la separación de servicios de notificación y recomendación reduce el acoplamiento funcional del flujo principal y habilita una arquitectura más extensible.

No obstante, la investigación también demuestra que dicha modernización no representa una mejora automática o universal. El enfoque distribuido incorpora mayores exigencias de observabilidad, despliegue, coordinación entre componentes y gestión de fallas, por lo que su adopción sólo se justifica cuando estos costos se encuentran alineados con la necesidad real de escalar, integrar nuevas capacidades o aislar procesos secundarios del núcleo transaccional.

En términos académicos, la principal contribución del trabajo consiste en proponer una ruta de modernización analizada, modelada y evaluable, aplicable a escenarios semejantes de sistemas heredados. Si bien la validación cuantitativa quedó parcialmente delimitada por el carácter controlado de la simulación, el estudio deja establecidos los escenarios de prueba, las métricas, la arquitectura objetivo y los mecanismos de observación necesarios para una fase posterior de verificación empírica.

Como proyección, se recomienda profundizar la ejecución experimental de las pruebas de carga y de falla controlada, ampliar la evidencia comparativa mediante dashboards y trazas consolidadas, e incorporar nuevos servicios o integraciones que permitan evaluar la escalabilidad selectiva y la resiliencia en contextos más cercanos a producción.

Referencias bibliográficas

[1] BASS, L., CLEMENTS, P. y KAZMAN, R. Software architecture in practice. 4ª ed. Addison-Wesley, 2021.

[2] BURNS, B., BEDA, J. y HIGHTOWER, K. Kubernetes: Up and running. 3ª ed. O'Reilly Media, 2022.

[3] FORD, N., RICHARDS, M., PARSONS, R. y KUA, P. Building evolutionary architectures. O'Reilly Media, 2017.

[4] FOWLER, M. Microservices: A definition of this new architectural term [en línea]. MartinFowler.com, 25 de marzo de 2014. <https://martinfowler.com/articles/microservices.html>.

[5] HERNÁNDEZ SAMPIERI, R., FERNÁNDEZ COLLADO, C. y BAPTISTA LUCIO, P. Metodología de la investigación. 6ª ed. McGraw-Hill, 2014.

[6] KIM, G., HUMBLE, J., DEBOIS, P. y WILLIS, J. The DevOps handbook. 2ª ed. IT Revolution, 2021.

[7] KLEPPMANN, M. Designing data-intensive applications. O'Reilly Media, 2017.

[8] NEWMAN, S. Building microservices. 2ª ed. O'Reilly Media, 2021.

[9] RICHARDS, M. Software architecture patterns. O'Reilly Media, 2015.