UNIVERSIDAD NACIONAL DE SAN JUAN

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

DEPARTAMENTO DE INFORMÁTICA

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

MIGRACIÓN DE UN SISTEMA MONOLÍTICO A UNA ARQUITECTURA DE MICROSERVICIOS. ÁMBITO: FINTECH.

TRABAJO DE FIN DE CARRERA: INFORME TÉCNICO DE PRACTICA PROFESIONAL

ALUMNO: FRANCISCO OLDECOP - REG. N°: XXXX

ASESORA: MG. MARÍA INÉS LUND

SAN JUAN 2026

Agradecimientos

Hoy deseo expresar mi sincero agradecimiento a quienes acompañaron mi formación y dejaron una marca profunda en mi recorrido.
A Javier P., Mariano A., Franco V., Franco G., y Daniel B., por brindarme un entorno exigente y profesional, donde pude aprender sin temor al error.
Sin su confianza y apoyo, mi experiencia habría sido considerablemente más limitada.
A Juan T., por su disposición y ayuda.

Debo también realizar la mención correspondiente a mi familia.
Imposible decir cuánto me han ayudado a lo largo de este camino.
Gracias, debo decir primero, incluso en aquellos momentos en que mis prioridades no resultaron del todo claras para ellos.
Aun así, valoro su ayuda.

El lugar de honor pertenece a los miembros del grupo de los sábados y a quienes forman parte de la orden, cuya compañía es una parte esencial de quien soy.
Este recorrido fue intenso, y sin su acompañamiento difícilmente habría llegado hasta acá.
Bajo cualquier circunstancia volvería a elegirlos como equipo.
Otra mención especial corresponde a Martín, MotS. Que nos une a todos, aunque ya no está con nosotros.
WES. A la institución que me abrió sus puertas, por confiar en mi potencial y desafiarme día a día.
Al final del trayecto, deseo recordar también a A., Mauricio y José, cuya contribución a mi experiencia profesional fue verdaderamente inolvidable.
Indudablemente, todo lo que podía decirse está dicho.

Resumen

Este informe documenta el desarrollo de una práctica profesional en WES, una consultora tecnológica de San Juan, Argentina, especializada en desarrollo de software y transformación digital. El proyecto consistió en ayudar a un cliente del sector fintech —que está en plena expansión— a modernizar su plataforma tecnológica. Específicamente, se trabajó en migrar su sistema monolítico hacia una arquitectura de microservicios para mejorar la escalabilidad, para que sea más fácil de mantener y se integre sin problemas con nuevos servicios financieros.

El trabajo inició analizando el sistema existente para entender cómo estaba construido y dónde estaban los principales problemas. Después se investigó qué tecnologías y patrones arquitectónicos podrían funcionar mejor para este caso particular. Con eso, se diseñó una estructura modular basada en dominios funcionales y se desarrolló un prototipo que sirviera como base para empezar la migración. También se definieron estándares de código, guías de documentación y criterios de calidad, porque el cliente necesitaba mejorar sus prácticas técnicas en general.

Los resultados concretos fueron: un diseño de arquitectura modular aprobado por los ejecutivos del cliente, un boilerplate técnico reutilizable para futuros desarrollos, y las bases para completar la migración más adelante. Esta experiencia permitió aplicar en un proyecto real conocimientos sobre ingeniería de software, arquitecturas distribuidas y metodologías ágiles, consolidando habilidades técnicas en un entorno profesional real.

Palabras clave: arquitectura de microservicios, sistema monolítico, FinTech, modernización de software.

Abstract

This report documents the development of a professional internship at WES, a technology consulting firm in San Juan, Argentina, specialized in software development and digital transformation. The project consisted of helping a fintech sector client—which is in full expansion—to modernize its technology platform. Specifically, the work focused on migrating its monolithic system towards a microservices architecture so it could scale better, be easier to maintain, and integrate seamlessly with new financial services.

The work started by analyzing the existing system to understand how it was built and where the main problems were. Afterwards, research was conducted on which technologies and architectural patterns could work best for this particular case. With that, a modular structure was designed based on functional domains and a prototype was developed to serve as a foundation to begin the migration. Code standards, documentation guidelines, and quality criteria were also defined, because the client needed to improve its technical practices in general.

The concrete results were: a modular architecture design approved by the client's executives, a reusable technical boilerplate for future developments, and the foundations to complete the migration later on. This experience made it possible to apply knowledge about software engineering, distributed architectures, and agile methodologies in a real project, consolidating technical skills in an actual professional environment.

Keywords: microservices architecture, monolithic system, FinTech, software modernization.

Capítulo 1 - Introducción

1.1. Contexto de la práctica profesional

La práctica profesional se llevó a cabo en WES, una consultora argentina de tecnología con más de diez años de trayectoria en el sector IT. La empresa tiene su sede principal en San Juan y trabaja con equipos distribuidos en distintas regiones de Latinoamérica. WES se especializa en desarrollo de software a medida, consultoría en transformación digital y gestión de talento tecnológico. Su enfoque se centra en acompañar a las organizaciones en procesos de modernización mediante metodologías ágiles y soluciones tecnológicas adaptadas a cada contexto.

El proyecto donde participé fue para una empresa nacional del sector logístico que se encuentra expandiéndose hacia el ámbito fintech. La organización buscaba modernizar su plataforma tecnológica, originalmente desarrollada como un sistema monolítico, hacia una arquitectura más flexible y escalable. Esta necesidad surgió por la incorporación de nuevos productos financieros, mayores requisitos de seguridad y trazabilidad, y la urgencia de mejorar la integración con servicios externos y plataformas bancarias.

WES fue contratada para liderar el proceso de transformación tecnológica, brindando soporte técnico, consultoría especializada y desarrollo. El rol desempeñado fue bajo la modalidad de staff augmentation[1], lo que implicó trabajar integrado en el equipo del cliente mientras mantenía vinculación con WES. El mismo combinó análisis funcional, diseño arquitectónico, desarrollo backend y asesoramiento en mejora de prácticas de desarrollo.

Uno de los primeros desafíos no fue técnico sino organizacional: el equipo utilizaba canales de comunicación informales (WhatsApp, Slack personal, mensajes directos) que dificultan el seguimiento de requerimientos y acuerdos. Antes de avanzar con el trabajo técnico, fue necesario ordenar los mecanismos de comunicación y establecer procesos más formales de documentación.

[1] **Staff Augmentation**: modalidad de contratación en la que una empresa incorpora temporalmente profesionales externos para complementar su equipo interno, con el fin de cubrir necesidades específicas de habilidades, aumentar la capacidad operativa o acelerar proyectos sin ampliar permanentemente su plantilla.

El entorno de trabajo presentaba una infraestructura en transición, con sistemas legado de considerable complejidad y documentación escasa o inexistente. Había componentes en proceso de migración conviviendo con tecnologías antiguas aún en uso. Este contexto, si bien complejo, representó una oportunidad valiosa para abordar problemáticas reales de modernización de software. Como parte de la práctica, se participó activamente en la planificación y ejecución de la estrategia de migración hacia microservicios, aplicando buenas prácticas de ingeniería y criterios de calidad que en muchos casos el equipo no había implementado previamente.

1.2. Propósito del informe

Este informe tiene como propósito documentar, analizar y reflexionar sobre las actividades desarrolladas durante la práctica profesional realizada en la empresa **WES**, en el marco de un proyecto de **migración tecnológica** orientado a transformar un sistema monolítico en una arquitectura basada en microservicios.

Este informe describe el **contexto del proyecto**, los **objetivos propuestos**, las **tareas ejecutadas** y los **resultados alcanzados**, destacando la aplicación práctica de los conocimientos adquiridos durante la carrera de **Licenciatura en Ciencias de la Computación**. Se procura evidenciar el proceso de integración entre la formación académica y las exigencias del ámbito profesional, mostrando cómo las competencias técnicas, analíticas y metodológicas se ponen en práctica en un entorno real de desarrollo de software.

Además de la descripción técnica, el documento pretende ofrecer una **reflexión crítica y profesional** sobre la experiencia, identificando los principales desafíos afrontados, los aprendizajes obtenidos y las oportunidades de mejora detectadas durante la participación en el proyecto.

1.3. Descripción general del proyecto desarrollado

La práctica profesional se centró en **las fases iniciales del proceso de migración**, abarcando el relevamiento técnico y funcional del sistema legado, la investigación de tecnologías y frameworks adecuados, el diseño de una arquitectura modular orientada a dominios, y la implementación de un **prototipo base (boilerplate)** que sirviera como punto de partida para la futura separación en microservicios.

También se colaboró en la definición de **estándares de codificación, lineamientos de documentación y**

criterios de calidad del software, con el propósito de fortalecer la estructura técnica y metodológica del equipo de desarrollo del cliente.

El proyecto se desarrolló bajo un enfoque **ágil**, con iteraciones de planificación, desarrollo y revisión continua, y una interacción constante entre los equipos técnicos de WES y del cliente. A lo largo del proceso se buscó no solo generar resultados técnicos tangibles, sino también promover una **cultura de mejora continua** y de adopción de prácticas modernas de desarrollo y gestión del software.

1.4. Objetivos

Se plantean dos tipos de objetivos: los académicos que son los que se busca lograr con la realización de la práctica profesional, y los objetivos propios del trabajo a realizar, que se busca satisfacer con el desarrollo del mismo.

Objetivos académicos

Elaborar un informe que documente, analice y reflexione sobre la experiencia adquirida durante la práctica profesional, evidenciando la aplicación de los conocimientos teóricos, prácticos y metodológicos de la carrera de **Licenciatura en Ciencias de la Computación** en un entorno laboral real, y evaluando el desarrollo de competencias técnicas y profesionales vinculadas al ejercicio de la disciplina.

- **Describir** el contexto institucional, las actividades realizadas y el rol desempeñado durante la práctica profesional.
- **Relacionar** las tareas desarrolladas con los contenidos y enfoques teóricos de la formación universitaria.
- **Identificar** las herramientas, metodologías y tecnologías utilizadas en el entorno de trabajo.
- **Reflexionar** sobre el proceso de aprendizaje y las habilidades adquiridas o fortalecidas a lo largo de la experiencia.
- **Evaluar** el grado de integración entre la formación académica y las exigencias del ámbito profesional.
- **Detectar** oportunidades de mejora personal y formativa a partir de la práctica.
- **Contribuir** a la consolidación del perfil profesional mediante la documentación técnica y analítica de la experiencia vivida.

Objetivos Laborales de la Práctica Profesional

Los objetivos planteados durante el período de práctica profesional se centraron en la aplicación de conocimientos técnicos y el desarrollo de habilidades blandas en un entorno de trabajo real, con foco en el proyecto de migración de sistemas en el ámbito FinTech.

Específicamente, los objetivos laborales alcanzados incluyeron:

- **Modularización de sistemas**: Colaborar en la descomposición de un sistema monolítico heredado, identificando dominios funcionales claros para su futura conversión en microservicios.
- **Planificación de migraciones**: Participar en las etapas iniciales de investigación y diseño para la migración de datos y funcionalidades, anticipando dificultades técnicas y organizativas.
- **Diseño de arquitecturas escalables**: Aplicar principios de diseño para proponer arquitecturas basadas en microservicios que garanticen escalabilidad, mantenibilidad y seguridad en un entorno de información crítica.
- **Integración de plataformas**: Implementar estándares de comunicación (como APIs REST y OAuth) para asegurar la interoperabilidad segura con plataformas financieras externas.
- **Desarrollo de habilidades transversales**: Reforzar la capacidad de comunicación efectiva, trabajo en equipo y liderazgo en un ambiente distribuido y de alta complejidad.
- **Gestión de prioridades**: Aprender a gestionar tareas y generar valor en contextos de incertidumbre, adaptándose a las exigencias de calidad y los desafíos de una organización tradicional en transformación digital.

1.5. Estructura del informe

Este informe se encuentra organizado en ocho capítulos principales, con el objetivo de presentar de manera clara, progresiva y fundamentada el desarrollo de la práctica profesional, el marco institucional en el que se llevó a cabo y los resultados obtenidos.

- **Capítulo 1 – Introducción:** Expone el contexto general de la práctica, el propósito del informe, una descripción global del proyecto, los objetivos perseguidos y la estructura general del documento.

- **Capítulo 2 – Marco institucional:** Describe la empresa FinTech en la que se realizó la práctica profesional, su estructura organizacional y el funcionamiento del área de tecnología, así como el rol desempeñado por el estudiante dentro del equipo de trabajo.

- **Capítulo 3 – Marco teórico y tecnológico:** Desarrolla los fundamentos conceptuales y técnicos relacionados con la arquitectura de microservicios, comparándola con los sistemas monolíticos, y detalla las tecnologías, herramientas y patrones empleados en el proyecto.

- **Capítulo 4 – Planificación y alcance del trabajo:** Explica el diagnóstico inicial del sistema monolítico, los objetivos y alcances definidos para la práctica, las actividades planificadas, el cronograma y los recursos utilizados, junto con las consideraciones de confidencialidad establecidas con el cliente.

- **Capítulo 5 – Desarrollo del trabajo:** Presenta en detalle las etapas del proceso de migración, desde el análisis del sistema original hasta el diseño, implementación, integración y validación de los microservicios desarrollados.

- **Capítulo 6 – Resultados y conclusiones de la práctica profesional:** Analiza los resultados obtenidos tras la migración, beneficios alcanzados, limitaciones y desafíos enfrentados durante el proyecto.

- **Referencias bibliográficas:** Reúne las fuentes consultadas, normativas, manuales y artículos técnicos en formato APA (7ª edición).

Capítulo 2 - Marco Institucional

Este capítulo describe el contexto institucional en el que se desarrolló la práctica profesional, abarcando tanto el entorno de la empresa **WES**, donde se enmarca la experiencia, como el de su **cliente del sector FinTech**, para el cual se ejecutó el proyecto de migración tecnológica. La práctica se llevó a cabo bajo un rol de **consultoría y desarrollo backend**, colaborando con equipos técnicos de ambas organizaciones en un proceso de modernización de software.

Por razones de confidencialidad y en cumplimiento de los acuerdos firmados con la empresa, **la identidad del cliente será reservada a lo largo del informe**, manteniéndose únicamente una descripción general de su estructura y contexto tecnológico.

2.1. Descripción de WES

WES es una **consultora tecnológica argentina** con más de una década de trayectoria en el sector IT, especializada en la **provisión de servicios de desarrollo de software, consultoría tecnológica y gestión de talento en tecnología**. La empresa tiene su sede principal en la provincia de **San Juan, Argentina**, aunque sus equipos de trabajo se encuentran distribuidos en distintos puntos del país y de Latinoamérica, adoptando una modalidad de colaboración remota y flexible.

La filosofía de WES se basa en su carácter de empresa **“digital native”**, orientada a la innovación, la calidad técnica y la mejora continua de los procesos tecnológicos de sus clientes. Su enfoque combina la **implementación de soluciones digitales a medida** con la **consultoría estratégica**, ofreciendo acompañamiento integral en proyectos de transformación digital.

Entre sus principales líneas de negocio se destacan:

- **Software Factory / Desarrollo a medida:** diseño y desarrollo de sistemas multicanal, aplicaciones empresariales y soluciones digitales personalizadas.
- **Talent Solutions / Gestión de talento IT:** servicios de selección, incorporación y acompañamiento de profesionales tecnológicos bajo la modalidad de **staff augmentation** o **outsourcing especializado**.

- **Consultoría en transformación digital, BPM, RPA, BI, QA y DevOps:** optimización de procesos mediante automatización, aseguramiento de calidad, inteligencia de negocio y adopción de metodologías ágiles.

La empresa se distingue por su capacidad para **adaptarse a distintos contextos organizacionales**, integrándose en equipos de clientes con estructuras y niveles de madurez tecnológica diversos. Su objetivo es potenciar la productividad y la calidad de los proyectos mediante la aplicación de **buenas prácticas de ingeniería de software, arquitectura moderna y cultura DevOps**.

2.2. Descripción de la Empresa cliente

La práctica profesional se desarrolló como parte de un proyecto de un **cliente de WES perteneciente al sector FinTech**, una organización nacional en proceso de expansión que busca consolidarse como actor relevante dentro del ámbito de los servicios financieros digitales. Esta empresa combina operaciones tradicionales con soluciones tecnológicas orientadas a la **gestión electrónica de pagos, acreditaciones y operaciones digitales**, en un contexto de transformación organizacional hacia modelos más ágiles, escalables y orientados al dato.

El entorno tecnológico previo al proyecto presentaba una estructura basada en un **sistema monolítico heredado**, desarrollado a lo largo de varios años y adaptado a las necesidades inmediatas del negocio. Sin embargo, este enfoque genera **limitaciones de escalabilidad, mantenibilidad y calidad técnica**, las cuales se volvieron críticas frente al crecimiento del volumen de operaciones y la necesidad de incorporar nuevos productos financieros.

En este contexto, la empresa **contrató los servicios de consultoría y desarrollo de WES** para acompañar un proceso de modernización tecnológica integral. El proyecto tuvo como eje la **migración progresiva del sistema monolítico hacia una arquitectura modular y, posteriormente, de microservicios**, con el objetivo de mejorar la capacidad de evolución, la escalabilidad, la trazabilidad y la calidad del software.

2.3. Rol del estudiante en el equipo de trabajo

Durante el desarrollo de la práctica profesional, el estudiante se desempeñó como **miembro del equipo de staff IT permanente de WES**, integrado bajo la modalidad de **staff augmentation** en el equipo técnico

del cliente del sector FinTech. Esta posición implicó una participación activa en las **operaciones cotidianas del proyecto**, adaptándose a las dinámicas, herramientas y metodologías de trabajo del cliente, pero manteniendo al mismo tiempo una **vinculación orgánica con la estructura interna de WES**.

El rol principal consistió en **ejecutar las tareas técnicas y de análisis requeridas por el proyecto**, aplicando los conocimientos y la experiencia profesional del estudiante en **desarrollo backend, arquitectura de software y modernización de sistemas**. Entre las responsabilidades desempeñadas se incluyeron la participación en el diseño de componentes clave del sistema, la definición de estándares técnicos, la colaboración en decisiones de arquitectura, y el acompañamiento en la implementación de buenas prácticas de desarrollo y documentación.

Además del trabajo técnico, el rol incluyó una dimensión estratégica orientada a **velar por los intereses de WES dentro del entorno del cliente**, garantizando la calidad del servicio y la alineación de los objetivos técnicos con los acuerdos comerciales y operativos establecidos. En este sentido, el estudiante **reportó directamente a la gerencia de operaciones y comercial de WES**, informando sobre el progreso del proyecto, anticipando potenciales riesgos y proponiendo oportunidades de mejora o expansión de la colaboración con el cliente.

Capítulo 3 - Marco teórico y tecnológico

En este capítulo se elaboran algunos conceptos clave sobre arquitectura de software y microservicios empleados. Se presentan conceptos básicos de ingeniería de software e informática aplicada, como arquitectura de software, transición de sistemas monolíticos a microservicios, comunicación asíncrona entre componentes, definición de contratos de APIs e integración de metodologías de trabajo. Estos conceptos clarifican un poco el reto tecnológico a los que se enfrentó y estrategias adoptadas.

Además, se muestra como las normas aprendidas en ambientes académicos se materializan en entornos profesionales, donde la complejidad de problemas y las peticiones del mercado presionan para ofrecer respuestas fuertes y duraderas en el marco institucional que las circunda. El marco teórico, más que para definir términos, respalda la interpretación y valoración de la experiencia, brindándole coherencia y rigor al análisis de lo aprendido.

3.1. Arquitectura de software

La práctica del desarrollo de software ha evolucionado hasta el punto en que los desarrolladores pueden resolver problemas técnicos específicos mediante la búsqueda eficiente de soluciones existentes. Sin embargo, esta lógica no se traslada de manera directa al campo de la arquitectura. Como señalan Ford et al. (2021):

> “Software developers build outstanding skills in searching online for solutions to a current problem. For example, if they need to figure out how to configure a particular tool in their environment, expert use of Google finds the answer.
>
> But that’s not true for architects.
>
> For architects, many problems present unique challenges because they conflate the exact environment and circumstances of your organization (...)” (p. 01).

En un proyecto de esta naturaleza, no es posible simplemente googlear 'cómo migrar a microservicios' y seguir un tutorial. Cada decisión depende del contexto específico del cliente y sus necesidades técnicas y de negocio.

Bajo esta perspectiva, la arquitectura de software son todas las decisiones estructurales que tomamos sobre cómo se compone un sistema. Esto incluye cómo interactúan sus componentes y cómo se comportan. El objetivo es cumplir con los requisitos funcionales y de rendimiento de forma eficiente. Es el "esqueleto" del sistema, en torno al cual se construyen módulos, se definen responsabilidades y se garantiza la calidad del producto.

En la experiencia desarrollada, su propósito fue asegurar modularidad y separación de responsabilidades, interoperabilidad entre componentes, rendimiento adecuado, escalabilidad y seguridad. En el ejercicio profesional, la arquitectura orienta la implementación de microservicios, la integración de APIs, la definición de flujos de trabajo, los despliegues distribuidos, la generación de documentación técnica, las revisiones de código y la planificación técnica.

Los fundamentos arquitectónicos aplicados incluyen modularidad, separación de responsabilidades, cohesión, bajo acoplamiento, resiliencia ante cambios y reutilización de componentes. Estos principios permiten reducir la complejidad y facilitar el mantenimiento y la evolución tecnológica continua. La modularidad y la separación de responsabilidades favorecen la construcción de servicios autónomos; la cohesión y el bajo acoplamiento incrementan la flexibilidad; la resiliencia habilita la adaptación a nuevos requerimientos; y la reutilización promueve eficiencia y estandarización.

Durante el desarrollo del proyecto, la aplicación de estos principios sentó bases sólidas para una eventual transición hacia una arquitectura de microservicios. Esto permitió definir límites claros entre dominios funcionales, implementar APIs limpias, estructurar comunicación asincrónica ordenada, minimizar riesgos en módulos heredados y mejorar la eficiencia del equipo de desarrollo.

La arquitectura de software es, en esencia, el marco que nos permite tomar decisiones técnicas enfocadas en la calidad y en que el sistema pueda evolucionar según lo necesite la organización.

3.2. Principios de diseño de microservicios

En este proyecto, los microservicios se adoptaron como un patrón arquitectónico en el cual la aplicación existente se descompondrá en conjunto de servicios pequeños, independientes y desplegados por separado, cada uno dedicado a una función específica del dominio. A diferencia de los sistemas

monolíticos tradicionales, cada microservicio debe encapsular su propia lógica de negocio y comunicarse con otros servicios mediante APIs o eventos asíncronos, lo que aporta mayor flexibilidad.

Características a destacar:

Autonomía de despliegue: cada microservicio puede desarrollarse, probarse y mantenerse sin afectar directamente al resto del sistema.

Escalabilidad individual: permite ajustar recursos según la demanda específica de cada servicio, optimizando el uso de infraestructura.

Responsabilidad única: cada servicio cumple una función concreta dentro del dominio, facilitando la evolución del sistema.

Desacoplamiento: la comunicación mediante interfaces bien definidas reduce dependencias estructurales y favorece la evolución independiente de los componentes.

Resiliencia: fallos en un microservicio no necesariamente comprometen el funcionamiento global.

Tecnología heterogénea: cada servicio puede implementarse con el lenguaje, framework o base de datos más adecuado para su propósito.

Automatización operativa: la naturaleza distribuida exige integración continua, despliegue continuo, contenedores y orquestadores para garantizar estabilidad.

Este enfoque puede facilitar la construcción de sistemas escalables, modulares y resilientes, particularmente adecuados para entornos de alta complejidad y dinamismo. Este enfoque supone un cambio conceptual respecto al diseño monolítico tradicional, donde la ausencia de una estructura modular clara puede derivar en arquitecturas difíciles de mantener. Ford et al. (2021) advierten que una aplicación web compleja *"with event handlers wired directly to database calls and no modularity can be considered a Big Ball of Mud*[2] *architecture"* (p. 65). Los autores sostienen que este tipo de sistemas carecen de la característica esencial de la arquitectura —su estructura interna— y que, sin una

[2] Describe una arquitectura de software carente de estructura clara, caracterizada por alta complejidad, fuerte acoplamiento, ausencia de límites bien definidos y crecimiento desordenado a lo largo del tiempo. En este tipo de sistemas, las decisiones se toman de manera reactiva y pragmática, sin una visión arquitectónica coherente,

gobernanza adecuada, muchos sistemas tienden a degradarse hacia este estado, lo que obliga a los arquitectos a planificar explícitamente procesos de reestructuración (pp. 66).

En la práctica, no todas estas características se implementan de manera inmediata, pero la adopción de microservicios presenta un atractivo particular debido a la promesa de escalabilidad y a la necesidad de una estructura modular explícita que evite la degradación progresiva del sistema a la que esta sometido actualmente.

3.2.a. Diferencias: arquitectura monolítica vs arquitectura de microservicios

Un monolito es un patrón tradicional en el que se integran todos los componentes de una aplicación en una sola base de código y se despliegan completamente en forma conjunta. En contraposición a esto, la arquitectura de microservicios divide el sistema en servicios especializados, autónomos e independientes, creando diferencias técnicas y operativas sustanciales. Estas diferencias son presentadas en la tabla 1.

Tabla 1. Diferencias entre Arquitectura Monolítica y de Microservicios. Elaboración propia.

Característica	**Monolito**	**Microservicios**
Despliegue	cualquier cambio requiere desplegar toda la aplicación	cada servicio se actualiza de forma independiente, reduciendo riesgos
Escalabilidad	Se escala todo el sistema, pudiendo desperdiciar recursos	Se escala por servicio, según necesidad, optimizando el rendimiento
Mantenibilidad y modularidad	suelen tener menor separación de responsabilidades, dificultando cambios	cada servicio tiene una sola responsabilidad, impulsa el desacoplamiento y modularidad desde el diseño, facilitando mantenimiento y reutilización
Resiliencia	un fallo de un monolito puede secuestrar a toda la aplicación	un fallo en un microservicio queda limitado al servicio involucrado
Tecnología y heterogeneidad	utilizan un único stack tecnológico	soportan diversas tecnologías por servicio.
Integración y comunicación	se comunican entre sí internamente con llamadas directas	usan APIs o mensajería asíncrona, promoviendo acoplamiento débil y transparencia en las dependencias

Los microservicios son una evolución hacia sistemas más flexibles, escalables y resilientes que permiten modularidad, despliegue independiente e innovación tecnológica.

3.2.b. Desafíos de la arquitectura de microservicios

A pesar de sus ventajas, los sistemas basados en microservicios presentan complejidades significativas, como la gestión de la comunicación distribuida, la coordinación de transacciones entre microservicios, el monitoreo y trazabilidad de fallos, la implementación de seguridad consistente y la necesidad de herramientas de automatización y orquestación avanzadas. Estos desafíos requieren buena planificación, buenas prácticas de ingeniería y experiencia en operaciones distribuidas para garantizar que los beneficios se materialicen sin comprometer la estabilidad del sistema.

Estos desafíos requieren buena planificación, buenas prácticas de ingeniería y experiencia en operaciones distribuidas para garantizar que los beneficios se materialicen sin comprometer la estabilidad del sistema. En relación con esto, **Ford et al. (2021)** señalan que “Building services that model bounded contexts required a subtle but important change to the way architects designed distributed systems because now transactionality is a first-class architectural concern” (p. 26), subrayando que la gestión de transacciones deja de ser un aspecto técnico secundario para convertirse en una preocupación arquitectónica central.

- **Complejidad en la administración:** la delegación de servicios aumenta la complejidad en la coordinación, supervisión y gestión de dependencias, requiriendo herramientas de automatización y observación avanzadas.

- **Comunicación de servicios:** la *requirement* para comunicación mediante APIs o colas de mensajes implica riesgos de latencia, pérdida de mensajes o inconsistencias que deben abordarse mediante patrones de resiliencia y estrategias de manejo de errores.

- **Consistencia de datos:** descentralizar el almacenamiento hace que asegurar coherencia y consistencia de los datos se dificulte y haya que recurrir a técnicas de replicación, consistencia eventual o transacciones distribuidas.

- **Pruebas y depuración:** las pruebas de sistemas distribuidos es más difícil que las pruebas de un monolito, porque los errores pueden producirse del intento de comunicación entre servicios y no solo de la lógica interna de cada uno.

- **Sobrecarga operativa**: la necesidad de pipelines de CI/CD, contenedores, orquestadores y monitoreo aumenta la carga de trabajo operacional, demandando conocimientos avanzados de DevOps y administración de sistemas distribuidos.

3.3. Patrones y herramientas de integración

La migración de un sistema monolítico hacia una arquitectura basada en microservicios implica una **redefinición conceptual de los límites, dependencias y responsabilidades** dentro del software. En este proceso, los **patrones de diseño** y las **herramientas de integración** cumplen un rol central, al proporcionar estructuras y principios que orientan la modularización del código, la comunicación entre servicios y la preservación de la coherencia funcional en entornos distribuidos.

3.3.a. Patrones de diseño aplicados a la modularización y microservicios

En el diseño de sistemas complejos, el núcleo del software no reside en su infraestructura técnica, sino en su capacidad para resolver los problemas propios del dominio de negocio. Evans (2003) sostiene que la verdadera esencia del software está en modelar adecuadamente el dominio, ya que todas las demás decisiones técnicas cumplen un rol de soporte frente a ese propósito central. Cuando el dominio es complejo, comprenderlo exige un esfuerzo deliberado por parte de los desarrolladores, quienes deben profundizar en el conocimiento del negocio y perfeccionar sus habilidades de modelado. Sin embargo, en muchos proyectos esta tarea suele quedar relegada frente al atractivo de los desafíos puramente técnicos, lo que puede derivar en soluciones técnicamente sofisticadas pero conceptualmente desconectadas del problema real.

Desde esta perspectiva, la **arquitectura orientada por dominios** (*Domain-Driven Design*, DDD) es uno de los enfoques fundamentales en la transición hacia sistemas distribuidos. Este paradigma promueve la división del sistema en subdominios bien definidos, cada uno representando una parte coherente del modelo de negocio. Los microservicios se estructuran en torno a estos dominios, estableciendo fronteras claras de responsabilidad y reduciendo la interdependencia técnica y lógica. Esta organización favorece la mantenibilidad, escalabilidad y autonomía de cada componente del sistema.

Otros patrones y modelos arquitectónicos relevantes en esté contexto son:

El **patrón de adaptadores** (*Adapter Pattern*) se utiliza para interconectar componentes heterogéneos o con distintos niveles de modernización. Mediante capas de adaptación, los servicios nuevos pueden comunicarse con módulos preexistentes sin modificar su estructura interna. Este patrón resulta esencial en escenarios donde la modernización es progresiva, ya que permite compatibilizar tecnologías y modelos de datos disímiles sin comprometer la integridad del sistema.

El **patrón de fachada** (*Facade Pattern*) introduce una capa de abstracción que agrupa operaciones complejas bajo una interfaz unificada. Este enfoque simplifica la interacción entre componentes internos y facilita la exposición de funcionalidades a través de una API o *gateway*. En el contexto de arquitecturas distribuidas, las fachadas contribuyen a mejorar la legibilidad del sistema, reducir la complejidad cognitiva y establecer puntos de integración más controlados.

El **patrón de repositorio** (*Repository Pattern*) promueve la separación entre la lógica de negocio y la lógica de acceso a datos, al definir una capa intermedia encargada de gestionar las operaciones de persistencia. Esta abstracción permite reemplazar o distribuir los mecanismos de almacenamiento sin alterar el comportamiento del dominio, lo que resulta esencial para entornos donde los datos se distribuyen entre múltiples microservicios.

Por último, la **arquitectura orientada a eventos** (*Event-Driven Architecture*, EDA) se apoya en la comunicación asincrónica mediante publicación y suscripción de mensajes. Este patrón permite que los servicios respondan a eventos sin depender de llamadas directas, logrando un sistema más desacoplado y resiliente. Además, la propagación de eventos facilita la escalabilidad horizontal y el procesamiento distribuido, elementos clave en la operación de plataformas modernas.

3.3.b. Herramientas de integración en entornos Azure y Node.js

Las herramientas de integración proporcionan el soporte tecnológico necesario para implementar los patrones antes descritos, en entornos de desarrollo contemporáneos. En ecosistemas basados en **Microsoft Azure**[3] y **Node.js**[4], estas herramientas permiten estandarizar la comunicación, gestionar la exposición de servicios y orquestar la interacción entre microservicios.

En el entorno **Azure**, destacan:

[3] https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
[4] https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

- **Azure API Management (APIM[5]):** servicio que centraliza la **exposición, control y seguridad de APIs**. Facilita la aplicación de políticas de acceso, versionado de endpoints y monitoreo de consumo, consolidando la comunicación entre servicios internos y externos.

- **Azure Service Bus[6]:** plataforma de **mensajería empresarial** que implementa colas y tópicos bajo el modelo *publish/subscribe*. Su uso se asocia a la integración asincrónica entre servicios y a la implementación de flujos de trabajo desacoplados.

En el entorno **Node.js**, la integración y modularización del backend se apoya en herramientas ampliamente extendidas:

- **Express.js[7]:** framework minimalista orientado al diseño de **endpoints RESTful** y a la definición estructurada de rutas y controladores.

- **Axios[8]:** librería para la **comunicación HTTP sincrónica y asincrónica**, utilizada en la interacción entre microservicios y APIs externas.

- **Sequelize[9]:** ORM que implementa principios del **patrón de repositorio**, promoviendo la abstracción de la capa de datos y la independencia del modelo de negocio.

- **OpenAPI / Swagger[10]:** estándares para la **documentación y validación de contratos de servicio**, que aseguran interoperabilidad y consistencia entre equipos de desarrollo.

En conjunto, estos patrones y herramientas funcionan como la base conceptual y tecnológica sobre la cual se estructura la **integración de sistemas distribuidos**, garantizando cohesión interna, flexibilidad evolutiva y compatibilidad entre componentes desarrollados en distintos entornos o etapas de madurez tecnológica.

[5] https://learn.microsoft.com/en-us/azure/api-management/
[6] https://learn.microsoft.com/en-us/azure/service-bus-messaging/
[7] https://expressjs.com/
[8] https://axios-http.com/docs/intro
[9] https://sequelize.org/docs/v6/getting-started/
[10] https://swagger.io/specification/

3.4. Tecnologías utilizadas

La adopción de arquitecturas basadas en microservicios se apoya en un conjunto de tecnologías que permiten la **contenerización, orquestación, integración, comunicación y monitoreo** de sistemas distribuidos. Estas herramientas proporcionan la base técnica que posibilita la independencia, escalabilidad y resiliencia de los servicios. Entre las más representativas se destacan **Docker**[11], **Kubernetes**[12], **Grafana**[13], **RabbitMQ**[14] y **Redis**[15].

Docker, una plataforma de **contenerización** que permite empaquetar aplicaciones junto con sus dependencias en unidades portátiles llamadas contenedores. A diferencia de las máquinas virtuales, los contenedores comparten el núcleo del sistema operativo, lo que reduce el consumo de recursos y acelera el despliegue. En el contexto de los microservicios, Docker facilita la creación de entornos consistentes, reproducibles y aislados, favoreciendo la modularidad y el versionado independiente de cada componente.

Para orquestar todos estos contenedores se utilizó **Kubernetes**. Al tener varios microservicios corriendo, es necesario contar con un mecanismo de gestión: que reinicie los que fallen, que distribuya la carga, que escale automáticamente según la demanda. Kubernetes hace todo eso. En el proyecto, se configuraron para manejar el ciclo de vida completo de cada servicio, desde el deployment inicial hasta las actualizaciones sin downtime.

Grafana ofrece una herramienta de **observabilidad y visualización de métricas** utilizada para supervisar sistemas complejos. A través de la integración con fuentes de datos como Prometheus, InfluxDB o Azure Monitor, posibilita la construcción de tableros de control personalizados esenciales para la detección temprana de anomalías y la toma de decisiones basada en datos.

RabbitMQ: intermediario de mensajería (*message broker*) que implementa los principios de comunicación asincrónica entre sistemas. Basado en el protocolo AMQP (*Advanced Message Queuing Protocol*), RabbitMQ permite que los microservicios intercambien mensajes sin depender directamente unos de otros, lo que reduce el acoplamiento y mejora la tolerancia a fallos.

[11] https://docs.docker.com/
[12] https://kubernetes.io/docs/home/
[13] https://grafana.com/docs/
[14] https://www.rabbitmq.com/docs
[15] https://redis.io/docs/latest/

También se utilizó Redis como un **almacenamiento en memoria** orientado a estructuras de datos, ampliamente utilizado como caché, base de datos clave-valor o *message broker* ligero. En arquitecturas distribuidas, Redis permite mejorar la eficiencia de consultas frecuentes, manejar sesiones o implementar mecanismos de *rate limiting* y colas temporales.

3.5. Beneficios y desafíos de la migración en el contexto FinTech

La adopción de una arquitectura basada en microservicios dentro del sector **FinTech** representa una evolución tecnológica que busca mejorar la **agilidad, resiliencia y capacidad de innovación** de las plataformas financieras. Este tipo de organizaciones opera en un entorno altamente regulado, con requisitos estrictos en materia de **seguridad, trazabilidad, disponibilidad y cumplimiento normativo**, lo que convierte a la modernización arquitectónica en un desafío tanto técnico como estratégico.

3.5.a. Beneficios de la migración

Uno de los principales beneficios de la migración hacia microservicios es la **escalabilidad independiente** de cada componente del sistema. A diferencia de las arquitecturas monolíticas, donde el crecimiento de una funcionalidad implica escalar todo el sistema, los microservicios permiten ajustar recursos de manera granular, optimizando costos y rendimiento. Esta propiedad resulta fundamental en entornos FinTech, donde la demanda puede variar de forma abrupta debido a operaciones financieras, cierres contables o picos transaccionales.

La **resiliencia operativa** es otro beneficio clave. En una arquitectura distribuida, los fallos localizados no afectan necesariamente al resto del sistema, lo que contribuye a mantener la continuidad del servicio en contextos donde la disponibilidad es crítica. A esto se suma la posibilidad de aplicar estrategias de **despliegue continuo (CI/CD)** y **rollback automatizado**, que reducen los tiempos y mejoran la estabilidad.

La **modularidad funcional** también promueve la **agilidad organizacional**. Cada microservicio puede ser desarrollado, desplegado y mantenido por equipos autónomos, facilitando la adopción de metodologías ágiles y la incorporación de nuevas funcionalidades sin impacto significativo en el núcleo del sistema. En empresas FinTech, donde la innovación constante y la velocidad de lanzamiento al mercado son factores competitivos, esta capacidad de evolución continua representa un valor estratégico.

Adicionalmente, la adopción de microservicios impulsa una mejor **observabilidad y control** del ecosistema tecnológico. Mediante el uso de herramientas de monitoreo, trazabilidad de eventos y auditorías centralizadas, las organizaciones pueden obtener una visión más clara de su operación, un aspecto crucial para garantizar el cumplimiento de normativas financieras y de protección de datos.

3.5.b. Desafíos de la migración

A pesar de sus ventajas, la migración hacia microservicios introduce una serie de desafíos conceptuales, técnicos y organizativos. El primero de ellos es la **complejidad en la gestión de la comunicación y los datos**. En un sistema monolítico, la interacción entre componentes se produce de manera interna y controlada, mientras que en una arquitectura distribuida requiere protocolos explícitos, mecanismos de mensajería y políticas de consistencia de datos. Esto conlleva la necesidad de adoptar estrategias como la **consistencia eventual**, el **saga pattern** o la **orquestación de transacciones distribuidas**.

Otro desafío relevante es el **incremento en los requisitos de infraestructura**. La gestión de múltiples servicios independientes demanda herramientas avanzadas de orquestación (como Kubernetes), balanceadores de carga, monitoreo centralizado y mecanismos de despliegue automatizado. Estas necesidades implican un esfuerzo adicional en planificación, administración y mantenimiento, así como una curva de aprendizaje significativa para los equipos técnicos.

La **seguridad y el cumplimiento normativo** adquieren una dimensión más compleja en este contexto. La descentralización del sistema implica proteger múltiples superficies de ataque, asegurar la comunicación entre servicios y garantizar la integridad de los datos en tránsito y en reposo. En el sector FinTech, donde rigen regulaciones como PCI DSS, ISO 27001 o normativas nacionales sobre protección de datos financieros, la arquitectura debe diseñarse considerando estos requisitos desde sus cimientos.

Finalmente, el proceso de migración presenta un **desafío organizacional**. La transición desde un modelo de desarrollo monolítico hacia un enfoque distribuido requiere una reestructuración de los procesos internos, la adopción de nuevas metodologías y una cultura orientada a la colaboración interdisciplinaria. Sin un adecuado acompañamiento en la gestión del cambio, las ventajas técnicas de los microservicios pueden verse opacadas por problemas humanos, como la coordinación entre equipos, diferencias en el nivel de conocimientos o falta de metodologías y procedimientos establecidos, entre otros.

Capítulo 4 - Planificación y alcance del trabajo

La planificación del trabajo a realizar se abordó inicialmente con una etapa de diagnóstico del sistema existente, entendiendo que no es posible proponer cambios sin antes conocer en detalle cómo funciona la aplicación preexistente. Al tratarse de un sistema monolítico en uso, con múltiples dependencias acumuladas en el tiempo, resultó necesario analizar la estructura completa del sistema, identificar sus principales limitaciones y comprender los procesos que soportaba.

El primer paso fue entonces, relevar y documentar el estado del sistema, problemas técnicos, vulnerabilidades y áreas a mejorar.

4.1 Diagnóstico del sistema Monolitico Existente

Como parte del equipo de WES, se llevó adelante un proceso de diagnóstico integral sobre el sistema monolítico legacy de la plataforma del cliente, previo al inicio de las tareas de modernización. El objetivo principal de esta etapa fue evaluar el estado actual de la solución, identificar riesgos, registrar vulnerabilidades y revelar oportunidades de mejora que permitirán definir un plan de trabajo realista, alineado a las necesidades del sistema.

El análisis abarcó aspectos técnicos y operativos del backend, así como prácticas de desarrollo o manejo de errores, buscando el nivel de madurez general del sistema. A continuación, se describen los principales hallazgos:

4.1.a Falta de estándares, buenas prácticas y calidad del código

El análisis preliminar evidenció una falta generalizada de criterios de estandarización en el código. La ausencia de un estilo consistente dificulta el mantenimiento y el diagnóstico de errores.

Entre los problemas más relevantes observados:

- Componentes clave (controladores, middleware, modelos) estructurados de manera inconsistente.
- Controladores implementados con distintos patrones y sin una separación clara entre el manejo HTTP y lógica de negocio.

- Middleware heterogéneo, donde algunos módulos agrupan múltiples responsabilidades, otros una sola y muchos de ellos están implementados parcialmente.
- Baja modularización y escasa reutilización de código. Se encontraron controladores de más de 2.000 líneas donde la lógica de pagos estaba mezclada con el envío de correos o el registro de logs, lo que hacía imposible modificar una parte sin romper la otra
- Implementación insuficiente de validaciones, de manejo de excepciones y de sanitización de datos.
- Mecanismos de seguridad aplicados de forma irregular.
- Presencia significativa de **code smells**[16], como duplicación de código, código muerto y métodos excesivamente grandes.

Estos aspectos no solo afectan la legibilidad del sistema, sino que también aumentan la probabilidad de errores difíciles de detectar y limitan la posibilidad de introducir mejoras sin riesgo.

4.1.b Estructura y organización del sistema

Otro hallazgo importante fue la falta de una estructura clara y consistente en la arquitectura del sistema. La organización actual no sigue un patrón establecido, afectando la escalabilidad, dificultando la navegación del código y aumentando la complejidad en tareas de depuración y expansión de funcionalidades.

Entre los problemas más relevantes se identificaron:

- Ausencia de una capa de servicios formalizada. Gran parte de la lógica de negocio está distribuida entre controladores y utilidades ad-hoc.
- Ruteo y controladores con responsabilidades mezcladas. Validaciones, lógica de negocios, operaciones de base de datos y manipulación de respuestas dentro de un mismo archivo, sin respetar el principio de responsabilidad única.
- Inconsistencia en la agrupación por módulos o dominios funcionales. Las funcionalidades no están organizadas de manera coherente, lo que complica su localización y comprensión.

[16] Término referido a indicios en el código que sugieren un diseño pobre o malas prácticas, no impiden funcionar pero dificultan el mantenimiento.

Estas características refuerzan la fragilidad del backend y limitan su capacidad de evolucionar hacia un entorno más escalable y mantenible.

4.1.c Manejo de Errores y registro de eventos

El manejo de errores es uno de los puntos más débiles del sistema. El diagnóstico mostró:

- Inexistencia de un handler global de errores, lo que impide capturar, clasificar y registrar las fallas de manera uniforme.
- Uso extensivo de errores sin contexto ni clasificación (*throw new Error*), dificultando la trazabilidad.
- Registros irrelevantes o dispersos, limitados a escrituras a la consola genéricas y sin niveles de severidad estandarizados (*info, warning, error*).
- Inconsistencia en los mensajes devueltos al frontend. A veces excesivamente técnicos y a veces demasiado genéricos, y algunos inclusive con detalles sensibles.
- Ausencia de un mecanismo de filtrado de información al exterior.

La falta de un sistema formal de logging y manejo de errores/respuestas impide monitorear el comportamiento del backend, dificulta la resolución de incidentes, limita la capacidad de auditar o investigar fallas en producción y expone el sistema a riesgos de filtrado de información o vulnerabilidades a atacantes externos.

4.1.d Debilidades de seguridad

Durante el análisis, se identificaron diversas vulnerabilidades y prácticas insuficientes en materia de seguridad, que afectan de forma directa a la integridad y resiliencia del backend. Estas debilidades no solo elevan el riesgo operativo, sino que también comprometen la exposición de datos sensibles, y el funcionamiento seguro de la plataforma.

Los principales hallazgos incluyen:

- Validaciones inconsistentes o incompletas sobre los datos de entrada: varios endpoints reciben información sin controles adecuados de tipo, formato o rango, lo que abre la puerta a inyecciones, *payloads* malformados o comportamientos inesperados.

- Sanitización irregular de parámetros. Algunos flujos utilizan la estructura de la solicitud sin sanitizar o validar, confiando en el ORM para realizar filtrado de inyecciones.
- Mecanismos de autenticación insuficientes, rudimentarios o inconsistentes.
- Respuestas de error potencialmente reveladoras. Algunos endpoints exponen detalles internos del sistema, estructura del modelo o información técnica que podría ser explotada por terceros malintencionados.

4.1.e Modelo de datos y uso del ORM[17]

El sistema utiliza Sequelize como ORM, pero su aplicación presenta limitaciones que impactan en la claridad, consistencia y confiabilidad del modelo de datos.

Entre los problemas identificados se encuentran:

- Mal manejo de transacciones. Ejecución de operaciones críticas sin envolverse en transacciones, o uso de transacciones demasiado abarcativas, generando bloqueos en el acceso a datos.
- Uso de consultas genéricas que recuperan tablas completas sin necesidad.
- Asociaciones entre modelos definidas de manera eventual e inconsistente, duplicando esfuerzos.
- No se implementan patrones estándar como repositorios o DTO[18], dejando el acceso a datos disperso, acoplado a controladores y utilidades.

4.2 Alcance de la práctica profesional

La práctica profesional desarrollada estuvo orientada a la evaluación del sistema existente, la investigación de alternativas de modernización, la elaboración de propuestas y el desarrollo de prototipos que sirvieran como base para la evolución tecnológica de la plataforma. El alcance incluye tanto actividades de diagnóstico como tareas de diseño, experimentación, documentación y desarrollo preliminar.

---

[17] **Object-Relational Mapping**: Componente de software que automatiza la conversión entre objetos de un lenguaje de programación y tablas de una base de datos relacional, facilitando el acceso, manipulación y gestión de datos sin escribir SQL manualmente.

[18] **Data Transfer Object**: Patrón que utiliza objetos simples para transportar datos entre capas o módulos de una aplicación, sin contener lógica de negocio, con el fin de reducir dependencias y optimizar la comunicación.

4.2.a Actividades planificadas

Al inicio de la práctica se estableció un conjunto de actividades orientadas a comprender el estado actual del sistema y explorar posibles soluciones para su mejora. Entre las principales tareas planificadas se incluyeron:

- Análisis técnico del sistema legacy, relevando arquitectura, componentes, flujos lógicos y principales problemáticas.
- Investigación y documentación de soluciones propuestas, incluyendo prácticas de desarrollo (linting, manejo de excepciones estandarizado, sanitización de entradas, estandarización de componentes), técnicas de diseño (patrones de uso) y actualizaciones tecnológicas (TypeScript, Node.js, ES Modules).
- Elaboración de prototipos preliminares y demostraciones técnicas, destinados a validar la viabilidad de las soluciones consideradas.
- Desarrollo de un prototipo funcional para producción, basado en el servicio de onboarding de usuarios.
- Diseño de arquitectura a nivel organizacional, incluyendo la propuesta de un entorno basado en microservicios conforme a las expectativas del cliente.
- Investigación de mecanismos de asincronismo, incluyendo la propuesta de un entorno basado en microservicios conforme a las expectativas del cliente.
- Presentación de una propuesta técnica a los equipos de desarrollo, infraestructura y gerencia.
- Reelaboración de la propuesta técnica en base al feedback recibido.
- Desarrollo de un *boilerplate* reutilizable y comienzo de la nueva implementación.

4.2.b Recursos y herramientas

Durante la realización de la práctica, se emplearon recursos técnicos y organizativos que permitieron llevar a cabo las tareas de análisis, experimentación y documentación. Entre las herramientas utilizadas se destacan:

- Herramientas técnicas: Node.js, VS Code, Postman, GIT, Docker, Sequelize, Husky, ESLint y SwaggerHub.

- Recursos organizativos: Documentación bajo los estándares de WES, uso de repositorios del cliente y trabajo en modalidad de investigación y desarrollo (I+D).
- Acceso a infraestructura del cliente: Repositorios GIT institucionales, ambientes de QA y disponibilidad limitada a documentación interna.

4.2.c Consideraciones de confidencialidad y propiedad intelectual

Dado que el proyecto involucra información sensible y sistemas productivos, la práctica profesional se desarrolló bajo condiciones estrictas de confidencialidad. En particular:

- El cliente estableció un acuerdo de confidencialidad (NDA) y restricciones de acceso a entornos sensibles.
- Se consideró confidencial toda la información relacionada con el proyecto de migración, incluyendo el código fuente, modelos y diseños, datos de usuarios, documentación interna y detalles operativos o de procesos internos.
- La propiedad intelectual del software analizado y producido; y de la documentación generada durante la práctica corresponde exclusivamente al cliente.

Estas consideraciones han sido respetadas para la redacción de este informe, asegurando el cumplimiento de los compromisos de confidencialidad y propiedad establecidos por la organización cliente.

Capítulo 5 - Desarrollo del trabajo

5.1 Identificación de módulos candidatos a microservicios

Por motivos de confidencialidad y en cumplimiento de los acuerdos establecidos con el cliente, en esté documento no se detallan los dominios funcionales específicos ni los componentes particulares de la aplicación analizada. No obstante, se describe el enfoque metodológico utilizado para identificar módulos candidatos a ser transformados en microservicios, así como los criterios técnicos y conceptuales que guiaron esté proceso.

Estos módulos se identificaron a partir del análisis de la documentación técnica y funcional generada durante la etapa de diagnóstico, complementada con instancias de intercambio con personal técnico y referentes del área de negocio. Esté trabajo conjunto permitió contrastar la estructura real del sistema con la forma en que la organización comprendía y utilizaba sus procesos, facilitando la detección de límites funcionales, responsabilidades compartidas y dependencias críticas entre componentes.

El análisis de la documentación incluyó la revisión de diagramas de arquitectura, flujos de datos, estructuras de base de datos y descripciones de procesos, con el objetivo de reconocer agrupaciones naturales de funcionalidades. Se prestó especial atención a la cohesión interna de los módulos, al grado de acoplamiento con otras partes del sistema y a la frecuencia de cambios asociados a cada componente, ya que estos factores resultan determinantes al evaluar la viabilidad de una futura separación en servicios independientes.

Las reuniones con el personal técnico permitieron validar hipótesis surgidas del análisis documental, aclarar decisiones históricas de diseño y comprender restricciones operativas no siempre evidentes en el código o la documentación. Por su parte, las conversaciones con referentes del negocio aportaron una perspectiva orientada a los procesos y objetivos organizacionales, lo que ayudó a identificar áreas funcionales con identidad propia dentro del sistema.

Para estructurar esté proceso se adoptó un enfoque basado en **arquitectura orientada por dominios**, comúnmente conocida como **Domain-Driven Design (DDD)**. Este paradigma propone modelar el software a partir de los dominios del negocio, definiendo límites claros —denominados bounded contexts— dentro de los cuales los conceptos y modelos mantienen coherencia. Cada dominio

representa un área funcional con responsabilidades específicas y puede evolucionar de manera relativamente independiente del resto del sistema.

El patrón arquitectónico **DDD** favorece la descomposición de sistemas monolíticos al proporcionar criterios conceptuales para delimitar servicios potenciales, priorizando la cohesión interna y reduciendo el acoplamiento externo. Bajo éste enfoque, la identificación de candidatos a microservicios no se basa únicamente en aspectos técnicos, sino también en la estructura del negocio, los flujos de información y la frecuencia de interacción entre áreas funcionales.

Aunque el enfoque DDD ofrece claridad conceptual, su aplicación real demandó múltiples iteraciones y discusiones internas, ya que los límites de dominio no estaban explícitamente definidos en la organización.

La delimitación de los contextos de DDD no fue inmediata; hubo debates intensos con los referentes de negocio sobre qué módulos debían ser independientes o parte del 'Core Transaccional'. Esté debate aún sigue en pie para algunos elementos del sistema.

A partir de este proceso, se logró establecer una propuesta de segmentación del sistema en dominios funcionales bien definidos, sentando las bases para la posterior planificación de la migración hacia una arquitectura de microservicios. Este enfoque ayudó a reducir la complejidad del sistema analizado, aunque implicó un esfuerzo inicial significativo.

5.2 Diseño de la nueva arquitectura

El diseño de la nueva arquitectura se basó en un enfoque de **microservicios organizados por dominios funcionales (**ver Imagen 1, pp. 35**)**, utilizando la división descrita en el inciso anterior, con el objetivo de mejorar la modularidad del sistema, reducir el acoplamiento entre componentes y facilitar su evolución futura. Esta propuesta arquitectónica adopta principios de comunicación orientada a eventos, priorizando operaciones atómicas y una orquestación mínima, apoyada en mecanismos de mensajería asincrónica. La estructura general busca equilibrar la necesidad de modernización con las limitaciones técnicas existentes, permitiendo una transición progresiva desde el sistema monolítico.

En éste modelo, cada microservicio se organiza en torno a un dominio funcional específico, encapsulando su lógica de negocio y limitando sus dependencias externas. La comunicación entre

servicios se realiza principalmente mediante eventos transmitidos a través de un **message broker**, en éste caso **RabbitMQ**, herramienta que permite gestionar colas de mensajería de manera confiable y desacoplada. Este enfoque favorece la resiliencia del sistema y reduce la dependencia de interacciones sincrónicas, mejorando la tolerancia a fallos y la escalabilidad.

El ingreso de solicitudes externas se canaliza a través de un **microservicio dedicado de interfaz**, responsable de recibir, clasificar y redirigir las peticiones según su naturaleza. Este componente actúa como punto de entrada único al sistema, separando las solicitudes que requieren respuesta inmediata de aquellas que pueden procesarse de forma diferida. Las solicitudes sincrónicas corresponden, en general, a consultas rápidas o a operaciones con requerimientos estrictos de tiempo de respuesta round-trip time), mientras que las solicitudes asincrónicas se vinculan a procesos que admiten mayor latencia o requieren la coordinación de múltiples servicios.

Las **solicitudes sincrónicas** se gestionan mediante interfaces tradicionales basadas en **APIs REST o GraphQL**, tecnologías que permiten consultas eficientes y estructuradas de información. Estas solicitudes se resuelven en el momento, retornando la respuesta directamente al solicitante una vez completado el procesamiento necesario.

Por otro lado, las **solicitudes asincrónicas** se transforman en eventos que se publican en colas de mensajería administradas por **RabbitMQ**. Para asegurar la trazabilidad del proceso, cada solicitud genera un identificador único que se transmite como parte de la metadata del evento. Esté identificador permite correlacionar las distintas etapas del procesamiento distribuido y vincular los resultados finales con la solicitud original.

Durante las pruebas iniciales se levantaron dudas sobre la complejidad operativa que implicaría su mantenimiento en el corto plazo. Estas dudas siguen vigentes y deberán evaluarse a futuro.

La gestión de solicitudes pendientes y de las respuestas generadas se apoya en el uso de **Redis**, un sistema de almacenamiento en memoria de alta velocidad que funciona como caché. Redis permite registrar el estado de cada solicitud asincrónica y almacenar temporalmente los resultados, facilitando su posterior recuperación por parte de los clientes y reduciendo la carga sobre otros componentes del sistema.

En cuanto a la persistencia de datos, por limitaciones técnicas y operativas, la base de datos principal se mantiene bajo un esquema monolítico. Sin embargo, para evitar accesos concurrentes desordenados y preservar la integridad de la información, se propone la creación de un **servicio dedicado** de **interfaz de base de datos**, encargado exclusivamente de procesar operaciones de lectura y escritura SQL. Este servicio centraliza la interacción con la base de datos, evitando la dispersión de lógica de acceso a datos entre múltiples microservicios y reduciendo riesgos asociados a inconsistencias o bloqueos.

Se puede observar la arquitectura propuesta en la Imagen 1 (pp. 35).

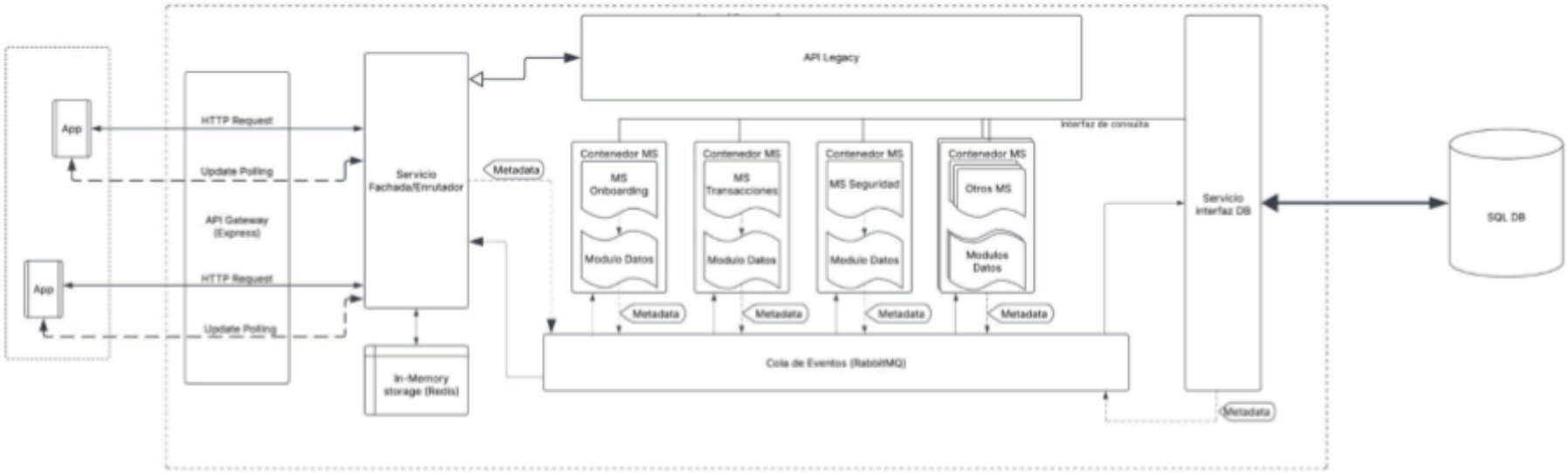

Imagen 1: Diseño de arquitectura de microservicios propuesto. Elaboración propia.

5.3 Implementación y despliegue de microservicios – Prueba de concepto

Con el objetivo de validar la arquitectura propuesta y evaluar su viabilidad técnica, se desarrolló una **prueba de concepto**, **(Proof of Concept, POC)** centrada en la implementación de un microservicio representativo. Esta etapa tuvo como finalidad definir una estructura base reutilizable, establecer criterios de desarrollo comunes y comprobar la integración de las tecnologías seleccionadas en un entorno controlado. El dominio elegido para esta primera implementación fue el de **gestión de usuarios**, particularmente los mecanismos asociados al **onboarding de nuevos usuarios**, debido a su claridad funcional, a su relativa independencia dentro del sistema y a la necesidad del negocio de contar con un nuevo mecanismo de onboarding.

La implementación del POC permitió construir un boilerplate general de microservicio, concebido como una plantilla de referencia para futuros desarrollos. Este boilerplate definió la organización interna del proyecto, separando responsabilidades en capas claramente delimitadas, tales como controladores, servicios de negocio, repositorios, modelos de datos y configuraciones de infraestructura. Esta estructura promovió la mantenibilidad del código, la reutilización de componentes y la incorporación de buenas prácticas de desarrollo desde las etapas iniciales.

Para el desarrollo se eligió trabajar con Node.js con TypeScript, elección que respondió a la necesidad de contar con tipado estático, mayor robustez en el desarrollo y mejor mantenibilidad del código a largo plazo. Como framework principal para la construcción de APIs se utilizó Express.js, debido a su flexibilidad y amplia adopción en entornos empresariales. Para la definición de esquemas y validación de datos se emplearon librerías especializadas que permitieron asegurar la consistencia de la información desde el ingreso al sistema.

En cuanto a las interfaces de comunicación, el microservicio incluyó tanto endpoints REST como una interfaz GraphQL, con el propósito de evaluar las ventajas y limitaciones de cada enfoque. Las APIs REST se orientaron a operaciones puntuales y de rápida resolución, mientras que GraphQL permitió explorar esquemas de consulta más flexibles, particularmente útiles en escenarios donde los consumidores del servicio requieren diferentes combinaciones de datos. Ambas interfaces coexistieron dentro del microservicio como parte del análisis comparativo de alternativas tecnológicas.

Adicionalmente, se diseñó una estructura básica para la gestión de eventos internos, estableciendo convenciones para la generación y el consumo de mensajes, aunque en esta etapa inicial no se integró un sistema de mensajería externo. Este diseño preliminar permitió definir contratos de eventos y preparar el microservicio para su futura integración en una arquitectura orientada a eventos.

El uso de Redis se contempló a nivel conceptual dentro del prototipo, particularmente en lo relativo a estrategias de cacheo y almacenamiento temporal de información. Sin embargo, dado que el microservicio se desarrolló de forma aislada durante esta fase, la implementación efectiva de mecanismos de caché de solicitudes o coordinación distribuida quedó fuera del alcance del POC.

Redis presenta un desafío técnico significativo para el equipo de infraestructura que debe instalar y configurar la instancia en Azure, pero presenta uno de los atractivos más amplios de las tecnologías propuestas. Queda pendiente evaluar cuándo y cómo se implementará.

Para la persistencia de datos, se implementó un prototipo de estructura de lectura y escritura (RW) utilizando Sequelize como ORM, permitiendo abstraer la interacción con la base de datos relacional. Esta capa facilitó la definición de modelos, validaciones y relaciones entre entidades, además de promover la separación entre la lógica de negocio y el acceso a datos. El objetivo principal fue evaluar la claridad del modelo de dominio y la mantenibilidad de la capa de persistencia, aunque no se abordaron todos los escenarios posibles.

En materia de seguridad de microservicios, los mecanismos de autenticación y autorización basados en JSON Web Tokens (JWT), así como técnicas de protección de credenciales mediante librerías de hashing criptográfico como bcrypt. También se implementaron validaciones de entrada, sanitización de datos y manejo estandarizado de errores para prevenir vulnerabilidades comunes, tales como inyecciones o accesos no autorizados.

Se intentó implementar una librería específica para Circuit Breaker, pero fue descartada rápidamente al notar que añadía una complejidad innecesaria para el volumen de tráfico actual del cliente. En un entorno preparado, este mecanismo permite detectar fallas recurrentes en dependencias externas y evitar la propagación de errores, contribuyendo a la estabilidad del sistema, pero resultó poco útil en la situación práctica actual. De momento, su inclusión forma parte del diseño base y está lista para futuras integraciones.

Finalmente, se definieron componentes reutilizables, tales como middleware de registro de eventos, validadores de esquemas, estructuras de respuesta estandarizadas, configuraciones compartidas y mecanismos de auditoría. Estos elementos sentaron las bases para la construcción de un ecosistema de microservicios coherente y mantenible.

Es importante señalar que, al tratarse de un microservicio implementado de manera aislada, esta etapa no incluyó aún la separación entre el servicio de interfaz de solicitudes externas, ni el servicio dedicado de acceso a base de datos, así como tampoco la integración con colas de mensajería o sistemas de cacheo de solicitudes. El enfoque se centró en validar la estructura interna del microservicio, la claridad del modelo de dominio y la viabilidad técnica del enfoque general propuesto.

La experiencia obtenida a partir de esta prueba de concepto permitió ajustar lineamientos de diseño, identificar oportunidades de mejora en la organización del código y consolidar un marco de referencia técnico para las siguientes etapas del proceso de migración. De esté modo, el POC constituyó un paso fundamental para reducir riesgos y fortalecer la coherencia arquitectónica del proyecto.

5.4 Pruebas, validación y control de calidad

Una vez completada la implementación inicial del microservicio y verificadas sus funcionalidades básicas en entornos locales, se inició el proceso de pruebas integrales y validación en conjunto con las áreas técnicas involucradas. Esta etapa tuvo como objetivo asegurar la estabilidad de la solución, su correcta integración con otros componentes del ecosistema y el cumplimiento de los criterios de calidad definidos previamente.

El primer paso consistió en el pasaje del desarrollo al entorno de Quality Assurance (QA), lo que requirió una coordinación estrecha con el área de Infraestructura. Durante esté proceso se trabajó en la configuración de un nuevo entorno de microservicios en la nube, utilizando Microsoft Azure como plataforma base. La contenerización de los servicios mediante Docker permitió estandarizar los despliegues y asegurar la consistencia entre los distintos entornos. A su vez, se utilizó Kubernetes como orquestador de contenedores, facilitando la administración de recursos, la escalabilidad y la supervisión de los servicios desplegados.

La configuración del entorno de QA implicó la definición de pipelines de despliegue, la gestión de variables de entorno, la asignación de recursos y la implementación de políticas básicas de seguridad y

acceso. Este trabajo conjunto permitió establecer una infraestructura que replicaba, en escala reducida, las condiciones del entorno productivo, lo que resultó fundamental para detectar problemas potenciales antes de su liberación final.

Posteriormente, se llevó a cabo el proceso de integración de la nueva API con la aplicación móvil existente. Esta etapa permitió validar la compatibilidad de las interfaces, los formatos de datos y los tiempos de respuesta. Durante la integración se realizaron ajustes menores en los contratos de comunicación para garantizar la coherencia entre los requerimientos del cliente y las respuestas del servicio. Esta instancia también facilitó la identificación de oportunidades de mejora en la definición de endpoints y en la estructura de las respuestas.

Las pruebas realizadas incluyeron escenarios funcionales, pruebas de regresión y evaluaciones de rendimiento básico. Se verificó el comportamiento del sistema ante diferentes condiciones de carga, la correcta gestión de errores y la estabilidad general del servicio. Durante este proceso se detectaron inconsistencias menos relacionadas con validaciones de datos, manejo de excepciones y sincronización de estados, las cuales fueron corregidas de manera iterativa.

El proceso de debugging desempeñó un rol central en esta etapa, apoyándose en herramientas de monitoreo, registros de eventos y trazabilidad de solicitudes. La instrumentación de logs estructurados permitió reconstruir flujos de ejecución y detectar puntos críticos en el procesamiento de información. Esta práctica contribuyó significativamente a la reducción de tiempos de resolución de incidentes y al fortalecimiento de la calidad general del servicio.

Una vez superadas las pruebas técnicas, se avanzó hacia el proceso de aceptación de usuario, donde representantes del área de negocio evaluaron el funcionamiento del sistema en función de los requerimientos definidos. Esta instancia permitió validar no solo aspectos técnicos, sino también la experiencia general de uso y la adecuación de la solución a los objetivos operativos. Las observaciones recibidas se tradujeron en ajustes menores que fueron incorporados antes del despliegue final.

Finalmente, se realizó el pasaje a producción siguiendo un enfoque controlado, priorizando la minimización de riesgos. El despliegue se llevó a cabo en coordinación con las áreas responsables, aplicando procedimientos de verificación posteriores a la liberación, para asegurar el correcto

funcionamiento del sistema en el entorno real. Esté proceso incluyó monitoreo continuo, revisión de métricas de desempeño y validaciones operativas iniciales.

La etapa de pruebas, validación y control de calidad permitió garantizar la confiabilidad de la solución propuesta, fortalecer la colaboración entre equipos técnicos y de negocio, y establecer una base sólida para futuras iteraciones del proceso de migración hacia una arquitectura de microservicios.

5.5 Presentación del modelo de arquitectura y planificación del roadmap de trabajo

Una vez definido el modelo de arquitectura propuesto y validados los primeros resultados del POC, se avanzó en la presentación formal de la propuesta a los distintos niveles de la organización. Estas instancias de comunicación tuvieron como objetivo alinear expectativas, identificar riesgos, recoger observaciones y asegurar que la estrategia técnica estuviera en sintonía con las prioridades del negocio.

Las presentaciones se realizaron de manera segmentada según el perfil de los participantes. Con el equipo de Desarrollo, el enfoque estuvo puesto en los aspectos técnicos del modelo de microservicios, incluyendo la separación por dominios, los patrones de diseño propuestos, los cambios en las prácticas de desarrollo y los desafíos asociados a la transición desde un sistema monolítico. Estas reuniones permitieron discutir preocupaciones relacionadas con la mantenibilidad del código, la complejidad operativa y la necesidad de establecer nuevos estándares de trabajo.

En paralelo, se llevaron a cabo encuentros con el área de Infraestructura, donde se abordaron los requerimientos necesarios para soportar una arquitectura distribuida, tales como la orquestación de contenedores, la observabilidad, la gestión de configuraciones y la automatización de despliegues. Estas conversaciones pusieron en evidencia que, si bien existía interés en evolucionar hacia microservicios, aún no se contaba con toda la infraestructura ni los procesos maduros necesarios para sostener esta arquitectura en el corto plazo.

A nivel de Negocio, las presentaciones se centraron en explicar cómo el nuevo modelo permitiría mejorar la escalabilidad del sistema y acelerar la incorporación de nuevas funcionalidades con menos riesgos a futuro. Sin embargo, también se destacó que una migración inmediata hacia microservicios implicaría un período de adaptación que podría afectar los tiempos de entrega. Esto resultó un punto de sensibilidad destacado, debido a la urgencia existente por ampliar las capacidades del producto.

Finalmente, en las instancias con Gerencia, se consolidó una visión estratégica que equilibrara los objetivos técnicos con las necesidades operativas. Es en estas reuniones que se llegó a la conclusión de que una transición directa hacia microservicios no resultaba viable en el contexto actual, tanto por limitaciones de infraestructura como por la necesidad de mantener la velocidad de desarrollo.

Como resultado de este análisis, se propuso un plan de trabajo a dos años, orientado a minimizar riesgos y permitir una adopción progresiva del nuevo paradigma arquitectónico.

Durante el primer año, el enfoque se centra en la migración del *codebase* existente hacia el nuevo *stack* tecnológico, incorporando las mejoras de diseño y prácticas definidas en el POC, pero manteniendo una estructura de monolito modular. Este enfoque no solo responde a una necesidad organizacional, sino que también encuentra respaldo en la literatura reciente. En el contexto de migraciones graduales desde un monolito hacia microservicios, Faustino et al. (2024) señalan que el paso intermedio de un monolito modular resulta ventajoso porque fomenta la separación de responsabilidades, permite visibilizar complejidades que deben abordarse antes de distribuir el sistema y facilita la toma de decisiones sobre la estrategia de migración (p. XX).

Este enfoque permite reorganizar el sistema bajo principios de arquitectura orientada por dominios, estableciendo límites claros entre módulos, reduciendo dependencias y mejorando la mantenibilidad del código, sin requerir aún una infraestructura completamente preparada para microservicios. Esta etapa también brinda el tiempo necesario para que el área de Infraestructura adapte sus herramientas, procesos y capacidades operativas a los nuevos requerimientos.

El diseño del monolito modular preserva la independencia conceptual y técnica de los dominios, aplicando principios de *Domain-Driven Design* para asegurar alta cohesión interna y contratos bien definidos entre módulos. De este modo, cada dominio evoluciona de manera relativamente autónoma, sentando las bases para una futura distribución del sistema.

Durante el segundo año, una vez consolidados los procesos de desarrollo y la infraestructura necesaria, el monolito modular puede dividirse progresivamente en microservicios independientes. Gracias a la separación previa por dominios, esta transición requiere principalmente la implementación de mecanismos de comunicación entre servicios —como mensajería, APIs y gestión de configuraciones—

sin necesidad de re-implementar la lógica funcional existente. Este enfoque reduce significativamente los riesgos técnicos y permite una adopción más controlada del nuevo modelo arquitectónico.

Esté roadmap equilibra la necesidad inmediata de evolución del producto con una estrategia de modernización sostenible a largo plazo, promoviendo una transformación gradual que prioriza la estabilidad operativa, el aprendizaje organizacional y la reducción de riesgos técnicos.

Capítulo 6 - Resultados y conclusiones de la práctica profesional

6.1 Resultados técnicos

La práctica profesional permitió alcanzar una serie de resultados técnicos que abarcan desde el entendimiento profundo del sistema existente hasta la definición de una estrategia concreta de modernización arquitectónica.

Estos avances se desarrollaron de manera progresiva, comenzando con el análisis del sistema monolítico, continuando con el diseño de una nueva arquitectura basada en dominios y culminando en la implementación de un microservicio POC que actualmente se encuentra en operación productiva.

En primer lugar, el proceso de relevamiento y diagnóstico inicial permitió obtener una visión clara de la estructura del sistema heredado, sus dependencias internas, los principales puntos de acoplamiento y las limitaciones técnicas que dificultaban su evolución.

Este análisis derivó en la generación de documentación técnica que no existía previamente de forma centralizada, incluyendo descripciones de módulos, flujos de información, contratos de integración y riesgos asociados a la arquitectura monolítica. Como resultado, se estableció una base de conocimiento compartida que facilitó la toma de decisiones posteriores.

A partir de esta comprensión inicial, se logró definir un modelo arquitectónico orientado a microservicios, basado en la separación por dominios funcionales y en el uso de patrones de diseño que favorecen la modularización y el desacoplamiento. Esté diseño introdujo lineamientos claros sobre la comunicación entre componentes, el manejo de operaciones sincrónicas y asincrónicas, la gestión de datos y la observabilidad del sistema.

Si bien la arquitectura completa aún no se implementa en su totalidad, su definición resulta un resultado técnico relevante, ya que establece un marco de referencia para futuras decisiones de desarrollo.

Un hito significativo de la práctica fue el desarrollo de un Proof of Concept (POC) que permitió validar la viabilidad técnica de la propuesta arquitectónica. Este microservicio, enfocado en un dominio específico

del sistema, implementó una estructura base reutilizable que incluye estándares de organización de código, mecanismos de seguridad, gestión de configuraciones, interfaces de comunicación REST y GraphQL, y una aproximación inicial a la persistencia de datos mediante ORM. Actualmente, este POC se encuentra en funcionamiento dentro de un entorno productivo, lo que demuestra la factibilidad del enfoque adoptado y su compatibilidad con el ecosistema tecnológico existente.

Además, la práctica permitió establecer un conjunto de lineamientos técnicos y buenas prácticas que sirven como referencia para futuros desarrollos. Entre ellos se incluyen convenciones de codificación, criterios de diseño modular, estrategias de manejo de errores, mecanismos de autenticación y autorización, y pautas para la documentación de servicios.

Estos elementos contribuyen a la estandarización del trabajo técnico y facilitan la incorporación de nuevos desarrolladores al proyecto.

Otro resultado técnico relevante fue la elaboración de un plan de migración a dos años, que propone una transición progresiva desde el sistema monolítico hacia una arquitectura distribuida.

Este plan introduce la adopción inicial de un monolito modular, organizado bajo principios de Domain-Driven Design, como paso intermedio hacia la futura implementación de microservicios. Desde el punto de vista técnico, esta estrategia reduce riesgos, permite una mejor gestión de dependencias y establece una base sólida para la posterior distribución del sistema.

La práctica profesional concluyó con el desarrollo de los primeros dominios dentro del nuevo monolito modular. No obstante, los resultados obtenidos hasta el momento evidencian avances concretos en términos de organización del código, claridad arquitectónica y preparación del entorno tecnológico para una evolución sostenida. El trabajo realizado no solo permitió validar conceptos teóricos en un contexto real, sino también sentar las bases técnicas necesarias para la continuidad del proceso de modernización del sistema.

6.2 Beneficios alcanzados para el negocio

Los avances técnicos y organizativos alcanzados durante la práctica profesional generaron beneficios concretos para el negocio, particularmente en términos de capacidad de respuesta, eficiencia operativa y mejora en la calidad de los entregables. Si bien la modernización del sistema es un proceso en curso, los

cambios introducidos en la forma de trabajo, en la coordinación entre equipos y en la infraestructura de despliegue ya produjeron impactos positivos en el desarrollo de nuevas funcionalidades.

Uno de los beneficios más visibles fue la implementación de un nuevo sistema de onboarding de usuarios, diseñado con un enfoque moderno tanto a nivel técnico como funcional.

Esté módulo permitió mejorar la experiencia de alta de nuevos clientes, reducir tiempos de procesamiento y ofrecer mayor flexibilidad para incorporar cambios futuros en requisitos regulatorios o comerciales. Desde la perspectiva del negocio, esto se traduce en una mayor capacidad para captar usuarios y adaptarse a nuevas demandas del mercado.

Por otro lado, la reorganización del trabajo técnico y la definición de lineamientos claros de desarrollo contribuyeron a una mayor agilidad en la implementación de nuevas funcionalidades. La estructuración del código, la estandarización de procesos y la mejora en la comunicación entre áreas técnicas y de negocio facilitaron la planificación de tareas, redujeron retrabajos y permitieron una estimación más precisa de tiempos de entrega. Esto impacta directamente en la capacidad del negocio para responder a oportunidades y ajustar su estrategia tecnológica.

Otro aspecto relevante fue la mejora en los procesos de despliegue en entornos de QA[19], UAT[20] y Producción.

La colaboración entre los equipos de desarrollo e infraestructura permitió establecer flujos de trabajo más ordenados, con mayor previsibilidad y menor riesgo de errores en los pasajes entre entornos. Estas mejoras no solo redujeron incidentes asociados a despliegues, sino que también aumentaron la confianza en cuanto a la estabilidad de las nuevas versiones del sistema.

Estos avances contribuyeron a fortalecer la relación entre los equipos técnicos y las áreas estratégicas de la organización, promoviendo una mayor alineación entre los objetivos tecnológicos y las necesidades del negocio. Aunque el proceso de modernización continúa, los resultados obtenidos hasta el momento

[19] **QA (Quality Assurance)**: entorno de pruebas destinado a la verificación técnica y funcional del software por parte del equipo de desarrollo o testing. En este ambiente se validan correcciones, integraciones y cumplimiento de requisitos antes de su liberación.

[20] **UAT (User Acceptance Testing)**: entorno de aceptación de usuario, donde usuarios finales o representantes del negocio validan que el sistema cumple con los requerimientos funcionales y operativos definidos, previo a su despliegue en producción.

sugieren que las mejoras implementadas pueden optimizar el desarrollo del software, sino que también generan un impacto positivo en la capacidad de la empresa para innovar y sostener su crecimiento.

6.3 Limitaciones y dificultades encontradas

El proyecto se desarrolló con restricciones importantes: presupuesto ajustado, equipo reducido y la presión constante de mantener el sistema operativo.

Estos son problemas comunes en migraciones de sistemas productivos. No podés simplemente apagar todo y reconstruirlo desde cero. Mientras diseñamos la nueva arquitectura, seguíamos corrigiendo bugs y agregando features al sistema viejo. Hubo semanas donde avanzamos muy poco en la migración porque tuvimos incidentes críticos en el entorno productivo.

El principal problema fue que éramos pocos para un proyecto de esta magnitud. La mayoría del equipo seguía manteniendo el sistema viejo mientras intentábamos diseñar el nuevo. Hubo semanas donde prácticamente no avanzamos porque surgían bugs críticos en producción que había que resolver ya.

Gran parte del trabajo inicial se llevó adelante con un equipo acotado, compatibilizando las tareas de modernización con las responsabilidades cotidianas del mantenimiento del sistema existente. Esta situación implicó extender los plazos de ejecución y priorizar cuidadosamente cada intervención, enfocándose en aquellas acciones que generen mayor impacto con el menor nivel de riesgo posible.

También representó un desafío adicional la necesidad de justificar inversiones en infraestructura, capacitación y nuevas herramientas tecnológicas. La adopción de arquitecturas modernas y prácticas de ingeniería de software requiere no solo recursos técnicos, sino también un compromiso organizacional que no siempre se encuentra disponible en etapas iniciales de un proceso de transformación.

Otro aspecto relevante fue la dificultad de introducir nuevas metodologías y buenas prácticas en un entorno de trabajo con hábitos fuertemente arraigados.

La adopción de estándares de documentación, prácticas de testing, controles de calidad, procedimientos de seguridad y nuevas tecnologías implicó un proceso gradual de concientización y alineación entre distintas áreas de la organización. Este desafío no se limitó a los niveles gerenciales, sino que también

involucró a los equipos técnicos internos, tanto de desarrollo como de infraestructura, cuyas dinámicas de trabajo estaban consolidadas en torno a modelos tradicionales.

Promover cambios en éste contexto requirió un esfuerzo sostenido de comunicación, demostración de beneficios concretos y generación de consensos, entendiendo que la transformación cultural suele ser tan compleja como la transformación tecnológica. En muchos casos, fue necesario equilibrar la incorporación de mejoras respetando los tiempos de adaptación de los equipos involucrados.

Por otro lado, el código heredado era un desastre. Encontramos funciones de más de 2000 líneas, recursividad donde no tenía sentido, dependencias circulares por todos lados. Hubo momentos donde se planteó seriamente reescribir todo desde cero (aunque al final del día esto no es viable en la situación actual).

A esto se sumó el uso de librerías obsoletas o en desuso, así como la ausencia total de documentación técnica, lo que incrementó el esfuerzo necesario para comprender el funcionamiento del sistema y evaluar el impacto de cualquier modificación.

Estas condiciones no solo elevan el riesgo de introducir errores durante el mantenimiento, sino que también limitan la capacidad de incorporar nuevas funcionalidades de forma ágil.

Se puso en evidencia a partir de estas limitaciones y dificultades que la migración hacia una arquitectura moderna no es únicamente un desafío técnico, sino también organizacional y cultural. No obstante, el reconocimiento explícito de estas barreras permitió diseñar estrategias de trabajo realistas, priorizando la sostenibilidad del proceso de transformación y sentando las bases para su continuidad en el mediano y largo plazo.

Referencias Bibliográficas

Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2021c). *Software Architecture: the Hard Parts: Modern Tradeoff Analysis for Distributed Architectures*. O'Reilly Media.

Richardson, C. R. (s. f.). *What are microservices?* microservices.io. Recuperado 9 de marzo de 2025, de https://microservices.io/

Casciaro, M., & Mammino, L. (2020). *Node. JS Design Patterns: Design and Implement Production-Grade Node. Js Applications Using Proven Patterns and Techniques, 3rd Edition*.

Evans, E. (2004). *Domain-driven design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Node.js — Introduction to Node.js. (s. f.). https://nodejs.org/en/learn

Pantaleo, G. (2012). *Calidad en el desarrollo de software*.

Microsoft. (s. f.). *Microservices Architecture Style - Azure Architecture Center*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Microsoft. (s. f.). *API Management documentation*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/api-management/

Microsoft. (s. f.). *Azure Service Bus Messaging documentation*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/service-bus-messaging/

Express - Node.js web application framework. (s. f.). https://expressjs.com/

Getting started | Axios Docs. (s. f.). https://axios-http.com/docs/intro

Getting started | Sequelize. (2025, 25 abril). https://sequelize.org/docs/v6/getting-started/

OpenAPI Specification - Version 3.1.0 | Swagger. (s. f.). https://swagger.io/specification/

Inc, D. (2026, 5 febrero). *Home*. Docker Documentation. https://docs.docker.com/

Kubernetes documentation. (s. f.). Kubernetes. https://kubernetes.io/docs/home/

Grafana Technical documentation | Grafana Labs. (s. f.). Grafana Labs. https://grafana.com/docs/

RabbitMQ Documentation | RabbitMQ. (s. f.). https://www.rabbitmq.com/docs

Redis Documentation. (s. f.). Docs. https://redis.io/docs/latest/