UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

MAESTRÍA EN

SOFTWARE CON MENCIÓN EN DISEÑO DE ARQUITECTURA DE SISTEMAS

RPC-SO-34-NO.778-2021

OPCIÓN DE TITULACIÓN:

PROYECTO DE TITULACIÓN CON COMPONENTES DE INVESTIGACIÓN Y/O DESARROLLO

TEMA:

DISEÑO DE UNA ARQUITECTURA DE SOFTWARE BASADA EN MICRO SERVICIOS PARA UNA EMPRESA DEDICADA A LA VENTA Y DISTRIBUCIÓN DE SERVICIOS DE ENERGÍA ELÉCTRICA

AUTOR(ES)

AMANDA ELIZABETH CABASCANGO GARCIA
MARCO PATRICIO CLAVIJO REASCOS

DIRECTOR:

GUSTAVO ERNESTO NAVAS RUILOVA

QUITO – ECUADOR
2025

Autor(es):

Amanda Elizabeth Cabascango Garcia
Ingeniero de Sistemas
Candidata a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
acabascango@est.ups.edu.ec

Marco Patricio Clavijo Reascos
Ingeniero de Sistemas
Candidato a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
mclavijor@est.ups.edu.ec

Dirigido por:

Gustavo Ernesto Navas Ruilova
Ingeniero Mecanico
Master Universitario en Ciencias y Tecnologias de la Computacion
gnavas@ups.edu.ec

Amanda Elizabeth Cabascango Garcia--Marco Patricio Clavijo Reascos
Diseño de una Arquitectura de Software basada en micro servicios para una empresa dedicada a la venta y distribución de servicios de energía eléctrica

DEDICATORIA

A mi madre Eufemia, a cuya luz y enseñanza son las que yo estoy tratando de seguir desde el cielo. Su amor y su fortaleza me han hecho no ceder ante el dolor y ante los tiempos difíciles. Esta acción de no ceder ante el dolor también es de ella.

A mi mamita Carmen, por el cariño que me ha demostrado a través de este tiempo, por estar siempre ahí, apoyando con la palabra para acompañarme y alentarme en cada uno de los momentos del proceso que he tenido que atravesar. La suya fue una compañía muy importante para mí. A mi hermano Misael y a mi padre: gracias por su apoyo total. A mi marido: gracias por la paciencia y el amor, por acompañarme en esta tesis y en la vida. Gracias por tu compañía en los momentos críticos y en cada uno de los logros que he podido conseguir. A ustedes, mi familia, dedico el trabajo con mucho respeto.

Amanda

AGRADECIMIENTO

Agradezco a toda mi familia, que me brindaron su apoyo y su fe constante en mis capacidades. A mamita Carmen por su cariño y consejos; a mi hermano por su apoyo incondicional que me brindo. A mi esposo compañero de vida y tesis, gracias por tu amor y apoyo incondicional durante esta etapa. Extiendo mi gratitud a mi tutor, por su guía, sus aportes y su compromiso durante el desarrollo de esta tesis. Su orientación fue fundamental para fortalecer mi formación y dar solidez a este proyecto.

Finalmente, agradezco a todas las personas que, de alguna manera, contribuyeron con su tiempo, consejo o apoyo para que este trabajo se haga realidad. A cada una de ellas, mi sincero reconocimiento.

Amanda

Tabla de Contenido

Índice de Tablas y Figuras

Diseño de una Arquitectura de Software basada en micro servicios para una empresa dedicada a la venta y distribución de servicios de energía eléctrica

Autor(es):

Amanda Elizabeth Cabascango Garcia
Marco Patricio Clavijo Reascos

Resumen

El proyecto desarrolla una propuesta de modernización tecnológica mediante la adopción de una arquitectura basada en microservicios para una empresa dedicada a la distribución y comercialización de energía eléctrica. El análisis de partida deja claro un problema crítico: los sistemas actuales funcionan como grandes bloques monolíticos basados en tecnologías propietarias que ya quedaron obsoletas. Esto no solo pone un freno a la escalabilidad y la interoperabilidad, sino que dispara los costos y nos deja expuestos ante fallos de seguridad. En pocas palabras, esta rigidez golpea directamente la eficiencia operativa e impide responder a tiempo cuando la demanda crece.

Para atender esta problemática, el proyecto plantea el diseño de una arquitectura objetivo basada en microservicios, apoyada en el ecosistema Java, con Spring Boot como tecnología de referencia. La propuesta se centra en definir una estructura modular que permita desacoplar los servicios críticos, mejorando su escalabilidad, disponibilidad y resiliencia a nivel conceptual. El diseño arquitectónico se formula bajo criterios de seguridad, gobernanza y escalabilidad horizontal, incorporando además lineamientos para la automatización de despliegues y la operación futura, sin abordar la implementación efectiva de la solución propuesta.

Para que esta transición sea ordenada y con el menor riesgo posible, se sugiere trabajar con metodologías ágiles como Scrum y adoptar una cultura DevOps que garantice la entrega y el monitoreo continuos. En el informe se detallan tanto el diseño técnico como las herramientas elegidas, destacando cómo este cambio permitirá a la organización adaptarse mejor, reducir dependencias y dejar el terreno listo para futuras expansiones tecnológicas.

Palabras clave:

Arquitectura, microservicios, migración de sistemas legados, Java, DevOps, Scrum, interoperabilidad, empresa eléctrica, infraestructura tecnológica

ABSTRACT

This project develops a technological modernization proposal through the adoption of a microservices-based architecture for a company dedicated to the distribution and sale of electricity. The initial analysis reveals a critical problem: current systems operate as large, monolithic blocks based on proprietary technologies that are now obsolete. This not only hinders scalability and interoperability but also drives up costs and leaves us vulnerable to security vulnerabilities. In short, this rigidity directly impacts operational efficiency and prevents us from responding promptly when demand grows.

To address this problem, the project proposes the design of a target architecture based on microservices, supported by the Java ecosystem, with Spring Boot as the reference technology. The proposal focuses on defining a modular structure that allows for the decoupling of critical services, improving their scalability, availability, and resilience at a conceptual level. The architectural design is formulated under criteria of security, governance, and horizontal scalability, also incorporating guidelines for the automation of deployments and future operation, without addressing the actual implementation of the proposed solution.

To ensure a smooth transition with minimal risk, it is recommended to use agile methodologies such as Scrum and adopt a DevOps culture that guarantees continuous delivery and monitoring. The report details both the technical design and the chosen tools, highlighting how this change will allow the organization to adapt more effectively, reduce dependencies, and pave the way for future technological expansions.

Keywords:

Architecture, microservices, legacy system migration, Java, DevOps, Scrum, interoperability, electric utility, technology infrastructure.

1. INTRODUCCIÓN

1.1. ANTECEDENTES

En las últimas décadas, el desarrollo de arquitecturas de software ha experimentado una transformación, evolucionando desde modelos monolíticos hacia sistemas distribuidos y, hacia arquitecturas basadas en microservicios (López, Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web). Este progreso responde a la necesidad de las organizaciones de incrementar la escalabilidad, la facilidad de mantenimiento y la capacidad de recuperación de sus sistemas, lo cual resulta relevante en sectores como el energético. (Lewis, 2024) Destacan que las arquitecturas de microservicios permiten fragmentar aplicaciones en módulos más pequeños, independientes y con capacidad de ser desplegados de manera autónoma, mejorando la agilidad en el desarrollo e implementación de soluciones.

En el sector de la distribución eléctrica, la transformación digital analizada por (Luis, 2016)ha dejado clara una premisa: es indispensable contar con sistemas lo suficientemente robustos para manejar grandes volúmenes de datos en tiempo real. La literatura reciente coincide en que optimizar estas arquitecturas no es solo una cuestión de eficiencia operativa, sino una herramienta clave para gestionar la información y mejorar la toma de decisiones.

Sin embargo, pasar a una arquitectura de microservicios no está exento de fricciones. El desafío principal radica en la complejidad de orquestar múltiples servicios y asegurar una comunicación fluida entre ellos. Como advierte (Newman S. , 2019) el uso de patrones de diseño incorrectos puede llevar a un acoplamiento excesivo, dificultando el mantenimiento futuro. A esto se suma el factor humano: la tecnología por sí sola no basta si no va acompañada de un cambio cultural hacia metodologías ágiles y DevOps (Kim G. a., 2024)

Mirando hacia América Latina, las empresas eléctricas han intentado modernizarse mediante arquitecturas distribuidas (Aghazadeh Ardebili) No obstante, la realidad

es que muchas chocan con la falta de infraestructura y la necesidad de capacitación, lo que ralentiza la transición. De ahí la importancia de diseñar soluciones que sean funcionales, pero también viables para organizaciones con recursos limitados.

En términos de resultados, los microservicios han facilitado la creación de herramientas innovadoras, desde facturación instantánea hasta monitoreo inteligente (Swift, 2024). Elevando la competitividad y la experiencia del cliente. Pero implementar esto con éxito exige una visión integral que alinee lo técnico con los objetivos del negocio. Precisamente, este proyecto busca cubrir esa brecha diseñando una arquitectura optimizada que integre escalabilidad, seguridad y las mejores prácticas, respondiendo específicamente a los retos de una distribuidora de energía eléctrica (Di Franceso, Lago, & Malavolta, 2018)

1.2. Objetivos

1.2.1. Objetivo General

Diseñar una arquitectura de software adaptable y escalable para una empresa dedicada a la venta y distribución de energía eléctrica, mejorando su capacidad de respuesta a cambios y nuevas funcionalidades.

1.2.2. Objetivos específicos

- Evaluar los requerimientos para la definición de la arquitectura a diseñar.
- Diseñar una arquitectura basada en microservicios para modularizar las funcionalidades del sistema, mejorando la escalabilidad, la mantenibilidad y la capacidad de respuesta.
- Definir los roles y responsabilidades para la gestión de microservicios, incluyendo estándares de documentación, seguimiento de dependencias y procesos de integración continua (CI/CD).

2. Determinación del Problema

La empresa, dedicada a la venta y distribución de energía eléctrica, enfrenta importantes desafíos a causa de la obsolescencia de su infraestructura tecnológica. Actualmente utiliza una arquitectura monolítica basada en herramientas propietarias de Oracle (Pérez, 2018), las cuales carecen de actualizaciones frecuentes y dificultan la integración con tecnologías modernas incurriendo en elevados costos de licencias. Esto ha tenido como consecuencia una infraestructura poco flexible que afecta a la seguridad y el mantenimiento, así como a la interoperabilidad. Y todo esto golpea sin piedad la agilidad, dejándola sin margen de maniobra frente a la creciente demanda del sector.

Ante este panorama amenaza la posición de la empresa. De no resolverse, se arriesga a un estancamiento en la innovación, fallas críticas en sus sistemas y serias desventajas competitivas (Pérez, 2018). Los sistemas heredados de los años 2000, al no haber sido diseñados para la flexibilidad y conectividad de hoy, representan un lastre. Aunado a esto, la falta de actualizaciones frecuentes los expone a graves vulnerabilidades y ciberataques (Kamlofsky, 2015). Estas condiciones, inevitablemente, impactarían de lleno en la eficiencia operativa, la percepción que tiene el cliente y la viabilidad del negocio en un sector tan competitivo.

Por lo tanto, se vuelve imperativo adoptar una arquitectura de software moderna, flexible y alineada con los estándares vigentes. Esta renovación debe garantizar alta disponibilidad y capacidad de escalar, facilitando a su vez la integración mediante tecnologías abiertas (Zapata, 2009) y reduciendo la dependencia de proveedores. Implementar esto optimizará los costos, fortalecerá la seguridad y asegurará la continuidad operativa.

Una transformación de este calibre no es un fin en sí mismo, sino una inversión que impulsa la innovación y consolida la posición de la empresa para futuros desafíos. Para manejar la transición, se emplearán metodologías ágiles como Scrum (Morandini, 2021) permitiendo un avance progresivo y con riesgos controlados,

mientras que la adopción de DevOps (Qumer Gill, 2018)) garantizará la fluidez en la integración y entrega continua.

De no resolver esta situación, la empresa está enfrentando desventajas competitivas, incrementos en costos operativos, fallas críticas en sus sistemas y estancamiento en innovación (Pérez, 2018). Los sistemas heredados de los años 2000 no fueron diseñados para la flexibilidad y conectividad que requiere el entorno actual. Además, la falta de actualizaciones frecuentes los hace vulnerables a ciberataques y fallos de seguridad (Kamlofsky, 2015). Estas condiciones impactarían la eficiencia operativa, la percepción del cliente y la sostenibilidad del negocio en un entorno altamente competitivo.

Por ello, es fundamental adoptar una arquitectura de software moderna, flexible y alineada con los estándares tecnológicos vigentes. Esta debe garantizar alta disponibilidad y escalabilidad, facilitar la interoperabilidad (Zapata, 2009) mediante tecnologías abiertas y reducir la dependencia de proveedores. Además, permitirá optimizar costos, mejorar la seguridad y asegurar la continuidad del negocio.

Esta transformación preparará a la empresa para futuros desafíos, impulsando la innovación, optimizando recursos y consolidando su posición en un mercado dinámico. La implementación de metodologías ágiles como Scrum (Morandini, 2021) permitirá una transición progresiva, optimizando recursos y reduciendo riesgos. Además, DevOps (Qumer Gill, 2018) facilitará la integración y entrega continua.

3. Alcance

El desarrollo de este proyecto se centra en el diseño meticuloso de una arquitectura de software sustentada en microservicios, pensada para una distribuidora de energía eléctrica. Nuestro objetivo principal es entregar un sistema que asegure una alta escalabilidad, una mantenibilidad sencilla y una capacidad de respuesta

superior, permitiendo a la organización adaptarse con agilidad a la creciente demanda del mercado (López, Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web)

La investigación se abordará en varias fases: primero, con un análisis profundo de los requerimientos específicos. Luego, con la definición técnica de la arquitectura de microservicios y la identificación de los roles necesarios para su futura implementación exitosa. A esto se suma la creación de lineamientos clave para la documentación, la gestión de dependencias y, de manera fundamental, la implementación de procesos de Integración y Entrega Continua (CI/CD) (Zampetti, 2021). Estas prácticas son esenciales para garantizar un desarrollo eficiente y cumplir con los más altos estándares de calidad del sector tecnológico (Duan, 2023)

Es crucial establecer el límite de este trabajo: el foco estará estrictamente en el diseño conceptual y técnico de la arquitectura, proponiendo metodologías y tecnologías idóneas (Castillo Torres, 2021). No obstante, la implementación práctica del sistema en un entorno de producción quedará fuera del alcance. Con este enfoque bien delimitado, entregaremos a la empresa una base sólida y estructurada para su transición hacia un sistema más flexible y preparado para el futuro.

4. Marco teórico referencial

Para diseñar la propuesta de arquitectura de microservicios, resulta fundamental dominar los conocimientos técnicos y funcionales del entorno de la empresa de energía. En consecuencia, dedicaremos este capítulo a desarrollar los conceptos teóricos necesarios. De esta manera, se proporcionará la base conceptual requerida para fundamentar la solución que se propondrá más adelante.

4.1. Fundamentos de la arquitectura de software

Es esencial abordar la arquitectura de software cuando se está diseñando una solución tecnológica, dado que esta disciplina establece cómo se estructuran y conectan los distintos componentes del sistema. Una buena arquitectura no solo divide un sistema en elementos discretos (módulos, clases, servicios u objetos), sino que también es responsable de definir el mecanismo de interacción entre ellos a través de interfaces claras, según la definición propuesta por (Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

Este enfoque resulta fundamental, pues garantiza que el sistema sea comprensible, escalable y, sobre todo, mantenible a lo largo del tiempo. Sumado a esta base, una arquitectura bien definida facilita simultáneamente la integración entre los equipos de trabajo, el cumplimiento de los requisitos funcionales y técnicos, y la puesta en marcha eficiente del sistema en la infraestructura física (servidores o centros de datos).

Contar con una arquitectura clara es todavía más importante en una empresa comercializadora de la energía eléctrica, puesto que por un lado admite procesar la intrincada clasificación de los procesos operativos, como así también la confiabilidad del sistema; y por otro lado permite ser más fácil adaptarse a los cambios del entorno tecnológico.

Por tal motivo, introducir y entender el concepto de arquitectura software en este propuesta no sólo es necesario, sino indispensable para poder construir una solución robusta, sostenible y alineada con el negocio.

4.1.1. Ciclo de desarrollo de la arquitectura de software

Al abordar el diseño de una arquitectura basada en microservicios, resulta fundamental comprender a fondo el ciclo de desarrollo arquitectónico. Este proceso es la clave para garantizar una base técnica sólida, directamente alineada con los objetivos del negocio y las necesidades operativas de la empresa. Una arquitectura bien definida no solo aumenta la eficiencia técnica del sistema, sino que permite gestionar la complejidad inherente a los servicios distribuidos, optimizando así la escalabilidad y facilitando la evolución del software a largo plazo.

El ciclo de desarrollo de la arquitectura de software comprende las siguientes etapas:

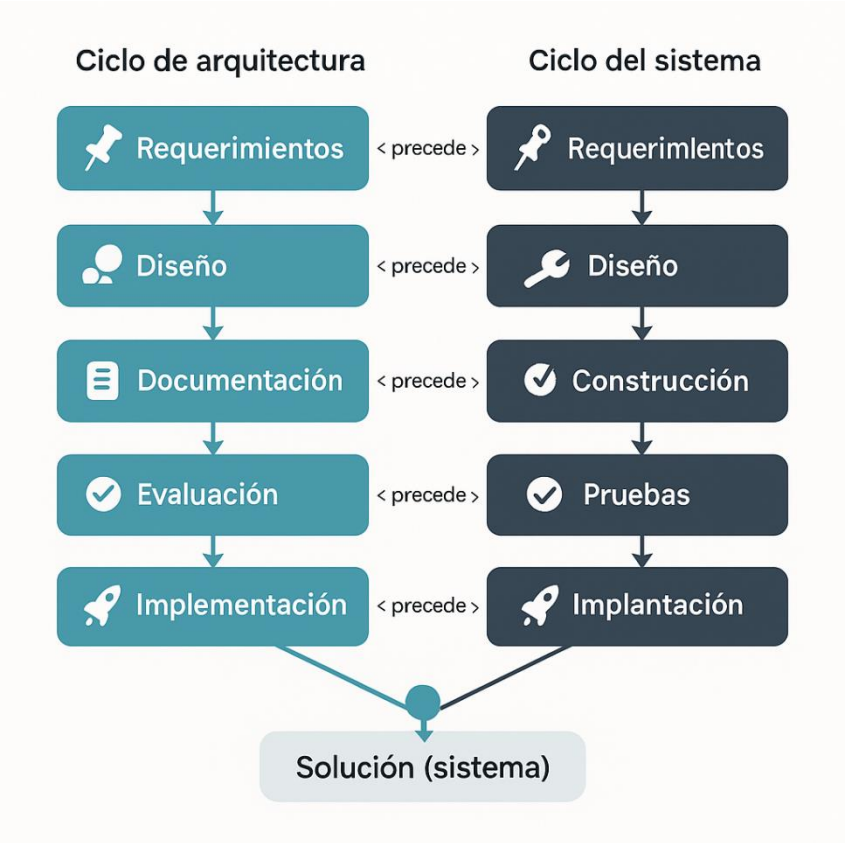

Figura 1. Ciclo de desarrollo de la arquitectura. Cervantes, H. (2016). Ilustración Basada de ARQUITECTURA DE SOFTWARE [Ilustración] (pp. 5).

I. Requerimientos de la arquitectura

Al abordar el diseño de una arquitectura basada en microservicios, resulta fundamental comprender a fondo el ciclo de desarrollo arquitectónico. Este proceso es la clave para garantizar una base técnica sólida, directamente alineada con los objetivos del negocio y las necesidades operativas de la empresa. Una arquitectura bien definida no solo aumenta la eficiencia técnica del sistema, sino que permite gestionar la complejidad inherente a los servicios distribuidos, optimizando así la escalabilidad y facilitando la evolución del software a largo plazo. (Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

II. Diseño de la arquitectura

La estructura del sistema se define en esta fase mediante decisiones clave de diseño. Es en este momento donde se aplican soluciones probadas, tales como patrones y tácticas arquitectónicas, junto con la selección de tecnologías específicas para microservicios. La importancia de esta etapa radica en que establece los cimientos técnicos para edificar el sistema, garantizando la respuesta a cada uno de los requerimientos identificados con anterioridad(Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

III. Documentación de la arquitectura

Una vez definido el diseño, es necesario comunicarlo adecuadamente a todos los actores involucrados en el proyecto, como desarrolladores, responsables de despliegue y líderes de equipo. La documentación se elabora a través de vistas arquitectónicas, que representan diferentes estructuras del sistema (lógicas, físicas o de ejecución), facilitando la comprensión del modelo propuesto y asegurando una implementación coherente (Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

IV. Evaluación de la arquitectura

Puesto que la arquitectura es fundamental para el éxito de cualquier sistema, se vuelve indispensable realizar una evaluación formal del diseño antes de iniciar la

codificación. Esta medida proactiva permite identificar riesgos y fallos potenciales de manera temprana, lo que reduce drásticamente los costos de corrección y garantiza que el sistema cumpla con todos los requisitos críticos establecidos desde el inicio. (Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

V. Implementación de la arquitectura

Una vez finalizada la fase de diseño, iniciamos la construcción del sistema. En este punto, es crucial mantener la coherencia y evitar a toda costa las desviaciones respecto a lo planeado. Debemos asegurarnos de que todos los servicios implementados se ajusten perfectamente al diseño arquitectónico establecido, lo cual es vital para sostener la cohesión y la consistencia del producto final. (Cervantes Maceda, Velasco Elizondo, & Castro Careaga, 2016).

4.2. Principios de diseño arquitectónico (modularidad, separación de responsabilidades, escalabilidad)

Al diseñar arquitecturas de software modernas, y especialmente si la elección son los microservicios, es vital aplicar principios guía que aseguren la calidad, la mantenibilidad y la evolución futura del sistema. Pilares esenciales como la modularidad, la separación de responsabilidades y la escalabilidad no son opcionales; son los verdaderos cimientos para construir soluciones que resulten eficientes, robustas y, sobre todo, adaptables.

Para el diseño de arquitecturas de microservicios, la modularidad actúa como un eje central. Este principio obliga a descomponer el sistema en componentes autónomos y claramente delimitados. El gran valor de esta separación es que cada pieza puede ser construida, validada y mantenida de forma totalmente independiente, lo que reduce sustancialmente el riesgo y la complejidad general del proyecto. En un contexto como el de una distribuidora eléctrica —con procesos vitales como la gestión de contratos, la facturación o la supervisión del consumo—, la modularidad es esencial: al tener estas funciones críticas aisladas, cualquier

cambio o mejora en una unidad se implementa sin amenazar la estabilidad del servicio completo. (Clemments, 2003)

Este principio refuerza la modularidad al exigir que cada componente se enfoque en una sola función o 'preocupación'. Esta disciplina técnica facilita enormemente la comprensión, la reutilización y la prueba de los componentes individuales. En un esquema de microservicios, esta filosofía se materializa en servicios pequeños y especializados, como un servicio de autenticación o de gestión de reclamos. Esta delimitación clara del propósito no solo aclara el código, sino que también permite delimitar las responsabilidades de cada equipo de desarrollo, mejorando la organización general del proyecto. (Rozanski & Woods, 2012)

Debido a que la demanda en el sector eléctrico implica manejar un flujo continuo y alto de transacciones en tiempo real, la escalabilidad es un factor decisivo. El diseño arquitectónico debe contemplar este principio para que sea posible incorporar nuevas funcionalidades o incrementar la capacidad de procesamiento sin tener que rehacer la totalidad del sistema. Esta flexibilidad es donde los microservicios demuestran su utilidad: permiten escalar granularmente, enfocándose únicamente en los servicios bajo presión, lo cual se traduce en una mejor optimización de recursos y una clara reducción de costos (Newman, 2015).

En conjunto, estos principios ofrecen una base sólida para diseñar una arquitectura de software eficiente, especialmente en entornos empresariales complejos y de alta demanda. Su aplicación adecuada asegura no solo una implementación técnica exitosa, sino también una alineación con los objetivos estratégicos de la organización.

4.3. Arquitecturas Monolíticas en Sistemas Críticos

Para comprender la evolución hacia una arquitectura basada en microservicios, es necesario partir del modelo tradicional que aún predomina en muchas organizaciones: la arquitectura monolítica. Este enfoque se caracteriza por integrar

todos los componentes del sistema interfaz de usuario, lógica de negocio, acceso a datos, entre otros en una única unidad ejecutable (Bass, Clements, & Kasman, 2012). En estos sistemas, los distintos módulos del software están fuertemente acoplados, lo que implica que cualquier cambio en una parte del código puede tener implicaciones en todo el sistema, dificultando tanto su mantenimiento como su evolución.

En el caso particular de la empresa dedicada a la venta y distribución de servicios de energía eléctrica, se identificó que su infraestructura tecnológica está construida sobre Oracle Forms y herramientas de la misma suite. Esta solución, si bien ha sido funcional durante 20 años, responde a una estructura monolítica, donde la lógica del negocio, la interfaz y el acceso a la base de datos están centralizados. Esto ha generado desafíos significativos relacionados con la flexibilidad del sistema, la incorporación de nuevas funcionalidades, y la respuesta ante cambios en los procesos operativos o en los requerimientos regulatorios del sector eléctrico.

El problema central que afecta al modelo de diseño monolítico es su incapacidad para realizar la administración de recursos desde una perspectiva granular: cuando se requiere que una funcionalidad específica administre las capacidades necesarias, el sistema la obliga a escalar todo el conjunto de la plataforma, lo cual puede resultar operativo incorrecto al ponerle un alto coste (Newman S. , Building Microservices: Designing Fine-Grained Systems, 2015). Este hecho tiene implicaciones operativas importantes en entornos exigentes a efectos de carga, como puede ser el caso de los módulos de atención al cliente y gestión de reclamos propios del sector energético. A la rigidez estructural ya mencionada se añade pues el riesgo inherente a depender de soluciones de tipo legacy (se habla de Oracle Forms) que quedan ancladas a navegadores y tecnologías obsoletas. Esto, no solo perjudica la adecuación a plataformas modernas, sino que crea serios problemas de vulnerabilidad a efectos de seguridad, mantenimiento y soporte. Todo esto lleva a pensar que se necesitan arquitecturas que sean menos frágiles y más modernas.

Dadas las condiciones de rigidez y de antigüedad del software, por tanto, es necesario proponer una arquitectura de software intrínsecamente más flexible,

más escalable y más cercana a los conceptos del desarrollo de software moderno. La arquitectura de los microservicios aparece al presentarse como la alternativa de arquitectura más adecuada para poder descomponer un sistema en servicios completamente autónomos mediante los cuales cada uno de ellos ejecuta la lógica de una funcionalidad específica. La propuesta simplifica de forma importante el desarrollo y la mantención y, además, permite dar una respuesta ágil para los requerimientos del entorno, escalar cada servicio de acuerdo a su demanda real, así como incorporar nuevas tecnologías sin comprometer la robustez general del sistema (Richards M. , 2015).

4.3.1. Acoplamiento y deuda técnica en sistemas legacy

Los sistemas legacy o heredados, como aquellos construidos sobre Oracle Forms, suelen presentar un alto nivel de acoplamiento entre sus componentes. Este acoplamiento se manifiesta cuando diferentes partes del sistema están tan interconectadas que realizan los cambios en una funcionalidad donde implica modificar varias áreas del código, generando riesgos de errores y retrabajos (Rozanski & Woods, 2012). En arquitecturas monolíticas, esta dependencia entre módulos es habitual, lo que puede complicar la implementación de mejoras o la incorporación de nuevas funcionalidades.

Además, este tipo de sistemas acumula lo que se conoce como deuda técnica, es decir, decisiones de desarrollo tomadas en el pasado que, aunque permitieron avanzar rápidamente en ese momento, ahora representan obstáculos técnicos y económicos. La deuda técnica se manifiesta en código obsoleto, ausencia de documentación, uso de tecnologías descontinuadas y dificultades para aplicar buenas prácticas de desarrollo moderno (Kruchten, Ozkaya, & Nord, 2012). En el caso específico de la empresa objeto de esta tesis, el uso prolongado de Oracle Forms ha generado una fuerte dependencia de herramientas antiguas que ya no son compatibles con los estándares actuales, agravando los efectos de esta deuda técnica.

4.3.2. Dificultades en mantenibilidad y evolución

Una consecuencia directa del acoplamiento y la deuda técnica en sistemas legacy es la pérdida progresiva de mantenibilidad. A medida que el sistema crece, mantenerlo actualizado o corregir errores se convierte en una tarea compleja y costosa, especialmente cuando no existen mecanismos claros de separación de responsabilidades entre componentes. Esto repercute negativamente en los tiempos de respuesta del área de tecnología, en la calidad del software y en la capacidad para adaptarse a nuevas necesidades del negocio.

En el caso de la empresa de servicios eléctricos estudiada, esta situación se agrava debido a la rigidez del sistema actual, el cual no solo presenta limitaciones técnicas para su escalabilidad, sino que también depende de entornos de ejecución obsoletos que dificultan su integración con otras plataformas. Como consecuencia, la evolución tecnológica se ve frenada, lo que afecta directamente la capacidad de innovación de la organización y su competitividad en un entorno cada vez más digitalizado.

Una arquitectura basada en microservicios permite mitigar estas dificultades al promover la división del sistema en servicios autónomos, con bajo acoplamiento y responsabilidades bien definidas. Este enfoque mejora la mantenibilidad, reduce la deuda técnica progresiva y facilita la evolución del sistema de forma controlada y continua (Newman S. , Building Microservices: Designing Fine-Grained Systems, 2015).

4.4. Arquitectura de Microservicios: Principios y Beneficios

En la actualidad, muchas organizaciones enfrentan desafíos tecnológicos derivados de la necesidad de ofrecer servicios más ágiles, escalables y adaptables. Ante esta realidad, la arquitectura de microservicios se posiciona como una solución moderna, especialmente útil en sectores con sistemas complejos y en constante evolución, como es el caso de las empresas dedicadas a la venta y distribución de

energía eléctrica. Esta arquitectura propone dividir las funcionalidades de un sistema en componentes pequeños, autónomos e independientes, que se comunican entre sí mediante APIs ligeras, mejorando así la mantenibilidad y el despliegue continuo de nuevas funcionalidades (Newman S. , Building Microservices: Designing Fine-Grained Systems, 2015).

A diferencia de las arquitecturas monolíticas tradicionales, en donde todas las operaciones del negocio se concentran en un solo bloque de código y una base de datos compartida, la arquitectura de microservicios permite una mayor descentralización y una distribución más flexible de las cargas de trabajo. Este enfoque no solo facilita el desarrollo paralelo por equipos distintos, sino que también mejora la tolerancia a fallos y la capacidad de respuesta ante cambios regulatorios o tecnológicos (Richardson, 2019).

4.4.1. Principios Fundamentales

Uno de los principios más relevantes de esta arquitectura es la descomposición funcional basada en el dominio del negocio, lo cual permite diseñar cada microservicio en torno a una funcionalidad concreta. Por ejemplo, en una empresa eléctrica, pueden identificarse microservicios como “medición”, “facturación”, “gestión de interrupciones” o “reclamos de clientes”, cada uno con su lógica propia y bases de datos independientes (Dragoni, y otros, 2017).

Además, se prioriza la independencia en el despliegue, lo que significa que cualquier servicio puede ser actualizado sin afectar al resto del sistema. Este principio está estrechamente vinculado con las prácticas DevOps y la automatización de pruebas e integración continua (Di Francesco, Lago, & Malavolta, 2018). También se promueve la comunicación ligera entre servicios, mediante protocolos como HTTP/REST o mensajería basada en eventos, lo que disminuye el acoplamiento y favorece la escalabilidad.

Por otro lado, la resiliencia se convierte en un atributo esencial. Gracias a patrones como circuit breaker o retry, es posible aislar fallos y mantener operativo el sistema global, incluso si uno de los microservicios presenta errores (Richardson, 2019).

Finalmente, la descentralización también se aplica al nivel de datos, lo que permite que cada microservicio gestione su propia persistencia y se adapte mejor a las necesidades del dominio que cubre.

Ejemplos de Microservicios en el Sector Energético

La aplicación de esta arquitectura en el sector energético no es solo viable, sino que resulta altamente beneficiosa. Algunas implementaciones concretas pueden observarse en:

Microservicio de Gestión de Contratos: Maneja el ciclo de vida de los contratos con clientes, desde su creación hasta su anulación o renovación, integrando validaciones legales y técnicas.

Estos casos no solo reflejan una división lógica del sistema, sino también la posibilidad de escalar cada servicio de forma independiente, según las necesidades operativas o estacionales de la empresa (Microsoft Azure Architecture Center , 2023).

Patrones de Diseño Aplicados a Microservicios

Para asegurar la eficiencia, mantenibilidad y escalabilidad del sistema, se aplican diversos patrones de diseño. Uno de los más comunes es el API Gateway, que actúa como punto de entrada unificado, controlando el acceso y enrutando solicitudes hacia los microservicios correspondientes. Esto es especialmente útil cuando el sistema sirve a múltiples tipos de clientes (internos y externos).

El patrón de Service Discovery también es clave, ya que permite a los servicios encontrar y comunicarse con otros de manera dinámica, una característica esencial en entornos donde los servicios pueden escalar o reiniciarse constantemente (IBM , 2021).

Otro patrón relevante es el de Saga, empleado para manejar transacciones distribuidas. En el contexto energético, se puede aplicar cuando se registra una nueva conexión: desde la solicitud del cliente hasta la activación física del

suministro, pasando por varias validaciones técnicas y administrativas (Richardson, 2019).

Adicionalmente, los patrones CQRS (Command Query Responsibility Segregation) y Event Sourcing son útiles para separar las operaciones de lectura y escritura, y para mantener un historial completo de eventos en sistemas donde la trazabilidad es clave, como en el monitoreo del consumo energético o en la auditoría de servicios prestados.

4.5. Beneficios de la arquitectura de microservicios

La arquitectura de microservicios es una opción sólida para construir sistemas flexibles, resilientes y escalables. Para empresas de energía eléctrica, esta arquitectura es útil para gestionar procesos complejos como la lectura de medidores inteligentes, la facturación dinámica, la gestión de redes y la atención al cliente multicanal.

La escalabilidad horizontal permite distribuir la carga entre varias instancias de un mismo servicio. Esto es útil en casos como el monitoreo en tiempo real o la facturación masiva, donde se puede escalar solo estos servicios para manejar picos de demanda (Villamizar, y otros, 2015).

Otro aspecto clave es la resiliencia ante fallos, ya que cada microservicio funciona de manera autónoma. En caso de que un componente falle, los demás pueden seguir operando, reduciendo así el impacto general sobre el sistema. Esto resulta vital en los sistemas energéticos, donde la continuidad operativa es un requisito crítico (Fazio, Celesti, Villari, & Puliafito, 2020).

Además, la arquitectura de microservicios impulsa la agilidad en el desarrollo y despliegue. Gracias a la separación de responsabilidades, los equipos de desarrollo pueden trabajar de manera paralela y autónoma en distintos servicios, facilitando

la entrega continua y reduciendo los tiempos de respuesta ante nuevas exigencias del negocio o cambios regulatorios (Balalaie , Heydarnoori, & Jamshidi, 2016).

En el ámbito tecnológico, esta arquitectura fomenta la heterogeneidad tecnológica, ya que cada servicio puede construirse con el lenguaje, framework o base de datos más adecuada a sus características. En el sector eléctrico, esto permite integrar tecnologías modernas como procesamiento de eventos en tiempo real, almacenamiento en la nube o inteligencia artificial para análisis predictivo.

La observabilidad del sistema es notoria, ya que la arquitectura de microservicios facilita la implementación de métricas, logs distribuidos y trazabilidad de eventos. Esto permite un monitoreo detallado del estado del sistema, lo cual es crucial en entornos operativos como las redes eléctricas inteligentes, donde es necesario detectar anomalías en tiempo real (Chen & Muhammad , 2014).

Característica	Arquitectura Monolítica	Arquitectura basada Microservicios
Escalabilidad	Escalado completo del sistema	Escalado por servicio o módulo
Tiempo de despliegue	Lento, requiere desplegar todo el sistema	Rápido, permite despliegues independientes
Tolerancia a fallos	Un fallo puede afectar todo el sistema	Aislamiento de fallos por microservicio
Tecnologías utilizadas	Homogéneas, restringidas a una misma tecnología	Heterogéneas, cada servicio puede usar tecnologías distintas
Mantenibilidad	Difícil en sistemas grandes	Más fácil por su modularidad
Adecuación a IoT	Limitada	Alta, facilita la integración con dispositivos y sensores
Monitoreo y trazabilidad	Centralizada, compleja de depurar	Distribuida y más detallada
Aplicación al sector energético	Menor adaptabilidad a procesos específicos	Alta adaptabilidad en diferentes áreas de la empresa.

Tabla 1. Cuadro comparativo entre las Arquitecturas Monolíticas y basada en Servicios. Autor: Ing. Amanda Cabascango

Conociendo los conceptos de las arquitecturas monolíticas y las arquitecturas basadas en microservicios, se elaboró una tabla comparativa (Tabla 1). En esta comparación, se identificaron los beneficios que se obtendrán al diseñar la nueva arquitectura basada en microservicios para una empresa eléctrica, considerando las necesidades específicas que deben satisfacerse durante el diseño de dicha propuesta.

4.6. Metodologías Ágiles en Arquitectura de Software

En el contexto de transformación digital y modernización de sistemas, las metodologías ágiles han cobrado especial relevancia en la arquitectura de software. Estas metodologías permiten abordar proyectos complejos de forma iterativa e incremental, mejorando la capacidad de adaptación ante cambios en los requerimientos, y facilitando la entrega continua de valor. La arquitectura de software, que históricamente ha sido percibida como una actividad previa al desarrollo, también se ha beneficiado del enfoque ágil al volverse evolutiva y colaborativa (Brown, 2019).

4.6.1. Aplicación de Scrum en proyectos de arquitectura evolutiva

Scrum es un marco de trabajo ágil que organiza el desarrollo en ciclos breves y repetitivos llamados sprints. Este enfoque permite abordar gradualmente la construcción de soluciones técnicas, permitiendo que la arquitectura se adapte a las necesidades del negocio y a la validación temprana de decisiones técnicas (Schwaber, 2020).

Scrum también facilita la integración de la arquitectura como parte del backlog, permitiendo priorizar decisiones técnicas junto con las funcionalidades de negocio, lo que garantiza un desarrollo armónico entre la visión técnica y la evolución del producto (Ford, 2017).

4.6.2. Gestión de deuda técnica y entregables incrementales

La deuda técnica representa decisiones subóptimas que se toman por conveniencia o urgencia, y que pueden impactar negativamente la calidad y evolución del software si no son atendidas. Las metodologías ágiles reconocen la inevitabilidad de esta deuda, pero proponen su gestión consciente y controlada. Bajo Scrum, es posible incluir tareas relacionadas con la refactorización, reestructuración o mejora técnica dentro del backlog, lo cual permite abordarlas en ciclos planificados (Kruchten P. N., 2012).

El desarrollo de entregables incrementales no solo se aplica a funcionalidades, sino también a componentes arquitectónicos. Esto permite validar en cada sprint aspectos clave como rendimiento, escalabilidad o acoplamiento entre servicios, reduciendo riesgos técnicos y facilitando decisiones fundamentadas (Martini, 2018).

4.6.3. Rol del arquitecto en equipos ágiles: facilitador técnico y garante de visión estratégica

En un entorno ágil, el arquitecto ya no actúa como una figura aislada que define el diseño completo al inicio del proyecto. Su rol evoluciona hacia el de facilitador técnico, apoyando al equipo en la toma de decisiones complejas, compartiendo conocimiento sobre patrones y tecnologías, y asegurando la reutilización de componentes (Richards 2020).

Además, el arquitecto es el guardián de la visión arquitectónica a largo plazo, alineando los esfuerzos técnicos con los objetivos estratégicos de la organización. En contextos de microservicios, donde múltiples equipos trabajan de manera descentralizada, esta figura resulta clave para mantener la coherencia técnica y garantizar que los servicios individuales contribuyan a todo el sistema (Wolpers, 2020).

4.6.4. Estrategias para refactorización o reescritura hacia microservicios

La transición desde Oracle Forms hacia arquitecturas modernas no puede plantearse como una simple reescritura, sino como un proceso estratégico y progresivo que contemple el rediseño del sistema bajo principios de modularidad, escalabilidad y resiliencia.

Una de las estrategias más reconocidas para esta transición es el Strangler Pattern, propuesta por (Newman S. , 2019), la cual consiste en desarrollar nuevos componentes como microservicios independientes que "rodean" gradualmente al sistema legacy, permitiendo su sustitución controlada sin interrumpir el servicio.

Otra alternativa es la reingeniería inversa, mediante la cual se documenta el sistema actual, se identifican los dominios funcionales y se priorizan los módulos a modernizar, especialmente aquellos de mayor valor para el negocio o con mayor rotación de cambios (Sommerville, 2020).

4.7. Seguridad en Microservicios para Entornos de Infraestructura Crítica

La adopción de arquitecturas basadas en microservicios en sectores como el energético, financiero, implica un replanteamiento profundo de los modelos de seguridad. A diferencia de las arquitecturas monolíticas, donde la seguridad puede centralizarse, en un ecosistema distribuido de servicios independientes se requiere aplicar enfoques descentralizados, escalables y resilientes para proteger la comunicación, el acceso y la integridad de cada componente del sistema.

Según ,la seguridad en microservicios debe concebirse como una capacidad transversal, que se aplica a nivel de infraestructura, servicio, red y datos, bajo principios como el mínimo privilegio, la autenticación federada, y el cero conocimiento previo entre servicios.

4.7.1. Gestión de Identidades y Control de Acceso a Nivel de Servicio (con LDAP)

Una práctica extendida en entornos empresariales es la integración del IdP con servidores LDAP (Lightweight Directory Access Protocol) o Active Directory, que siguen siendo ampliamente utilizados como fuente de autenticación interna. LDAP actúa como repositorio jerárquico de usuarios, contraseñas, grupos y políticas, y su integración con un IdP moderno permite extender las credenciales corporativas a servicios externos, sin necesidad de duplicar información (Auth0, 2023).

Por ejemplo, Auth0 ofrece un componente llamado LDAP Connector, el cual se instala en la red interna y actúa como puente entre la infraestructura local (LDAP/AD) y el servicio de autenticación en la nube. Este conector autentica a los usuarios directamente contra LDAP, pero entrega un token federado válido para consumir microservicios modernos (Auth0, 2023).

4.7.2. Modelos de seguridad Zero Trust, autenticación federada (OAuth2)

El enfoque de seguridad perimetral tradicional ha perdido eficacia frente a entornos altamente distribuidos. En su lugar, se ha adoptado el modelo Zero Trust, el cual asume que ninguna entidad dentro o fuera de la red es de confianza por defecto (Kindervag, 2010). En arquitecturas de microservicios, este enfoque se traduce en validar la identidad y los permisos de cada petición entre servicios, incluso si provienen de la misma red.

Uno de los pilares de Zero Trust es la autenticación federada, que permite delegar la autenticación en un proveedor de identidad externo (IdP) utilizando estándares como OAuth 2.0 y OpenID Connect. Estos mecanismos permiten emitir tokens firmados (como JWT) que contienen los claims de identidad y roles del usuario o sistema, y que pueden ser validados de forma autónoma por cada microservicio (Hardt, 2012)

Este modelo mejora la seguridad, facilita la escalabilidad y la interoperabilidad entre servicios, contenedores y dispositivos móviles, clave en plataformas críticas como la de una empresa energética.

4.7.3. Gestión de identidades y control de acceso a nivel de servicio

En un sistema basado en microservicios, el control de acceso debe implementarse a nivel de cada servicio, considerando tanto usuarios finales como otros servicios internos. Según (Fowler 2014),es recomendable adoptar el patrón de API Gateway con autenticación centralizada, que verifica las credenciales y delega las peticiones a los servicios internos, los cuales a su vez deben validar los permisos incluidos en los tokens.

Para una gestión eficaz de identidades, se recomienda el uso de Identity Providers (IdP) como Auth0, Keycloak o Azure AD, que permiten administrar usuarios, grupos, roles, políticas y flujos de autenticación desde una consola unificada. Estas plataformas también permiten aplicar autenticación multifactor (MFA) y políticas de acceso condicional, reforzando la seguridad de sistemas críticos.

Adicionalmente, los servicios pueden implementar mecanismos como el principio de mínimo privilegio, el uso de scopes en los tokens, y el control de acceso basado en atributos (ABAC), para asegurar que cada componente solo acceda a los recursos estrictamente necesarios.

4.7.4. Buenas prácticas en sectores regulados (energía, banca, salud)

Los entornos regulados exigen una capa adicional de controles y auditoría sobre los sistemas de software. En el caso del sector energético, por ejemplo, existen normas internacionales como NERC CIP en EE. UU., que obligan a implementar controles de acceso robustos, trazabilidad de operaciones y mecanismos de recuperación ante incidentes (NERC, 2023).

En el ámbito bancario, normativas como PCI DSS requieren el cifrado de datos sensibles en tránsito y en reposo, así como la separación de funciones entre servicios. En el sector salud, reglamentaciones como HIPAA exigen autenticación fuerte, control de sesiones y monitoreo de accesos.

(Siriwardena, Microservices Security in Action, 2019) recomiendan las siguientes buenas prácticas transversales:

- Asegurar la comunicación entre microservicios mediante mTLS.
- Usar cifrado de datos sensibles con algoritmos estándar como AES y RSA.

Estas prácticas garantizan que la arquitectura no solo sea funcional y escalable, sino también resiliente y conforme a los marcos regulatorios.

4.8. Orquestación y Observabilidad en Microservicios

Las arquitecturas de microservicios introducen un entorno altamente dinámico, distribuido y escalable, que requiere de mecanismos especializados para su despliegue, monitoreo y mantenimiento. En este contexto, tecnologías como Docker, Kubernetes y herramientas de observabilidad como Prometheus, Grafana o el stack ELK, se vuelven esenciales para asegurar el rendimiento, la resiliencia y la trazabilidad de los sistemas.

Además, en entornos de infraestructura crítica, como las empresas de energía eléctrica, es fundamental asegurar la alta disponibilidad, la capacidad de autoescalado, y la ejecución de despliegues seguros y sin interrupciones, dado que cualquier falla puede impactar directamente en la operación del negocio.

4.8.1. Contenerización (Docker) y orquestación con Kubernetes

Docker es una plataforma de contenerización que permite empaquetar aplicaciones y sus dependencias en contenedores ligeros, portables y reproducibles. Esta

tecnología es ampliamente utilizada en arquitecturas de microservicios, ya que facilita la estandarización del entorno de ejecución y permite un despliegue más ágil y consistente (Merkel, 2014).

La orquestación de contenedores se realiza comúnmente mediante Kubernetes, una plataforma de código abierto desarrollada por Google, que automatiza la implementación, el escalado y la gestión de aplicaciones contenerizadas. Kubernetes permite definir servicios, balanceadores de carga, políticas de tolerancia a fallos y reglas de autoscaling, todo a través de manifiestos declarativos .

Su arquitectura basada en clústeres permite mantener el estado deseado de los servicios distribuidos, facilitando su resiliencia, actualizaciones continuas y recuperación ante errores. En sistemas críticos, como los de gestión energética, esto permite mantener la operatividad continua incluso frente a fallos de hardware o errores humanos.

4.8.2. Observabilidad: logging distribuido

En un entorno de microservicios, donde las solicitudes pueden atravesar múltiples servicios y nodos, la observabilidad es clave para comprender el comportamiento del sistema. Esta se compone de tres pilares: logging, tracing y monitoring (Barrett, 2020).

Logging distribuido: permite centralizar los registros generados por cada servicio. El stack ELK (Elasticsearch, Logstash, Kibana) permite recolectar, indexar y visualizar logs de múltiples orígenes, facilitando el análisis de errores y la auditoría de eventos.

4.8.3. Alta disponibilidad, autoscaling y despliegues sin interrupciones (Blue-Green, Canary)

Una de las ventajas clave de Kubernetes es su capacidad para facilitar alta disponibilidad y despliegues progresivos, lo cual es fundamental en sistemas que no

pueden permitirse interrupciones. Kubernetes permite definir múltiples réplicas de un servicio, balancear la carga entre ellas y reemplazar nodos fallidos de forma automática.

Además, soporta mecanismos de autoescalado horizontal (HPA), que ajustan el número de réplicas en función del uso de CPU, memoria u otras métricas personalizadas (Hightower, Kubernetes: Up and Running: Dive into the Future of Infrastructure, 2017). Esto permite que el sistema responda dinámicamente a picos de demanda, garantizando niveles óptimos de servicio sin intervención manual.

Para realizar despliegues sin interrupciones, Kubernetes soporta estrategias como:

- Despliegue Blue-Green: donde dos versiones del servicio se ejecutan en paralelo, y el tráfico se redirige gradualmente hacia la nueva versión solo cuando ha sido validada. (Fowler M. , Blue Green Deployment, 2025)
- Despliegue Canary: que introduce nuevas versiones a un pequeño porcentaje de usuarios antes de implementarla completamente, reduciendo el riesgo de fallos. (Sato, 2025)

Estas estrategias son esenciales en sectores como el eléctrico, donde un error en producción puede afectar procesos de gestión de los diferentes procesos, facturación o distribución de energía.

4.9. Cultura DevOps y Automatización del Ciclo de Vida

El enfoque tradicional de desarrollo y operaciones como silos separados ha dado paso a una cultura DevOps, que promueve la colaboración continua entre desarrolladores, operadores, QA y seguridad para entregar software de forma más rápida, segura y confiable. Esta transformación es especialmente relevante en entornos críticos como los de energía eléctrica, donde el tiempo de respuesta ante incidentes, la estabilidad del sistema y la capacidad de escalar rápidamente son fundamentales.

DevOps no solo es una práctica técnica, sino también una mentalidad cultural que busca eliminar barreras, fomentar la automatización y aplicar principios de mejora continua en todo el ciclo de vida del software (Kim 2016).

4.9.1. Integración de CI/CD pipelines con Jenkins, GitLab CI,

Uno de los pilares de DevOps es la implementación de CI/CD (Integración Continua y Entrega/Despliegue Continuo). La integración continua (CI) permite validar automáticamente cada cambio en el código mediante pruebas y análisis estáticos, mientras que la entrega continua (CD) asegura que el software esté siempre en un estado desplegable.

Herramientas como Jenkins, GitLab CI/CD permiten construir pipelines que automatizan todo el flujo, desde el commit hasta el despliegue en producción. Jenkins destaca por su flexibilidad y extensa comunidad; GitLab CI ofrece integración nativa con el repositorio; y ArgoCD está orientado a despliegues continuos en entornos Kubernetes declarativos (Burns, 2022).

Este enfoque reduce errores humanos, acelera las entregas y permite mantener la consistencia entre entornos, lo cual es esencial para garantizar la disponibilidad de sistemas críticos.

4.9.2. Pruebas automatizadas a nivel unitario, integración y contratos

La automatización de pruebas es clave en DevOps, ya que garantiza la calidad del software sin frenar la velocidad de entrega. El enfoque moderno propone tres niveles principales de pruebas:

- Pruebas unitarias: validan funciones específicas de forma aislada. Herramientas: JUnit (Java), Jest (JavaScript), Pytest (Python)
- Pruebas de integración: validan el comportamiento entre módulos o servicios. Herramientas: Testcontainers, Postman CLI.

- Pruebas de contratos: aseguran que las APIs cumplan con lo que los consumidores esperan, incluso en entornos desacoplados. Herramientas como Pact permiten definir contratos entre servicios y verificar el cumplimiento mutuo (Fowler M. , 2020).

Estas pruebas automatizadas se integran en el pipeline CI/CD para validar cada cambio antes de llegar a producción. En sistemas distribuidos como los microservicios, esto es vital para mantener la estabilidad ante cambios frecuentes.

4.9.3. Infraestructura como código (IaC) con Terraform, Helm

La Infraestructura como Código (IaC) permite definir y gestionar recursos de infraestructura (redes, servidores, bases de datos, servicios en la nube) mediante archivos de configuración declarativos. Esto promueve la trazabilidad, repetibilidad y control de versiones sobre la infraestructura, alineándose con los principios de DevOps .

Terraform permite definir recursos en múltiples proveedores (AWS, Azure, GCP, etc.) mediante su lenguaje declarativo HCL.

Helm es el gestor de paquetes de Kubernetes, y permite desplegar aplicaciones complejas en clústeres mediante “charts” versionables, reutilizables y configurables.

Ambas herramientas permiten que el entorno de infraestructura se trate como parte del código fuente, facilitando su automatización, auditoría y despliegue sincronizado con los microservicios.

Este enfoque es esencial en empresas que necesitan controlar versiones de su entorno de infraestructura con el mismo rigor que su código de aplicación, especialmente en industrias reguladas.

4.10. Codificación Segura en Entornos de Desarrollo Distribuido

La codificación segura es una práctica fundamental para garantizar la protección de los sistemas de información frente a vulnerabilidades comunes que pueden ser explotadas por atacantes. En arquitecturas basadas en microservicios, donde los sistemas están altamente distribuidos y cada servicio expone interfaces accesibles, la importancia de aplicar principios de codificación segura desde la base del desarrollo se vuelve crítica.

La falta de controles de seguridad en el código fuente representa uno de los vectores más comunes de ciberataques, especialmente en sectores regulados como el energético, donde la disponibilidad, integridad y confidencialidad de los datos y servicios es prioritaria (OWASP, 2023).

4.10.1. Aplicación de estándares y guías de codificación segura (OWASP, CERT, ISO/IEC 27034)

Existen múltiples estándares y buenas prácticas de codificación segura diseñados para prevenir errores comunes de implementación. Entre los más reconocidos están:

- OWASP Secure Coding Practices, que proporciona directrices para proteger datos, manejar errores y aplicar validaciones de entrada/salida (OWASP, 2023).
- CERT Secure Coding Standards, promovido por el SEI de Carnegie Mellon, que proporciona lineamientos específicos por lenguaje (CERT, 2022).
- ISO/IEC 27034, que establece los principios del desarrollo de software seguro como parte de la gestión de seguridad de la información (ISO, 2011).

Estas guías promueven el diseño de sistemas seguros desde su concepción (secure by design), la validación de entrada/salida, el principio del mínimo privilegio y la protección de sesiones, criptografía y errores manejados de forma segura.

4.10.2. Integración de herramientas de análisis estático y dinámico en pipelines CI/CD

La automatización de la revisión de seguridad en el código fuente se ha vuelto una práctica indispensable en entornos DevOps. Las herramientas de análisis estático (SAST) permiten escanear el código sin ejecutarlo, identificando patrones inseguros, malas prácticas o uso indebido de funciones vulnerables (Hussain, 2021).

Herramientas como SonarQube, Semgrep, Checkmarx, o Fortify se pueden integrar directamente en pipelines CI/CD con Jenkins, GitLab o GitHub Actions, y detener despliegues si se detectan vulnerabilidades críticas.

Además, el uso de herramientas de análisis dinámico (DAST) como OWASP ZAP permite simular ataques en entornos controlados para validar el comportamiento de la aplicación en ejecución. Estas herramientas son especialmente útiles en entornos de staging donde se prueban endpoints expuestos de microservicios (Siriwardena, Microservices Security in Action, 2019).

4.10.3. Control de dependencias y librerías externas mediante gestión de vulnerabilidades

Los microservicios dependen ampliamente de librerías externas y paquetes de terceros. La inclusión de componentes no verificados puede introducir vulnerabilidades conocidas en el sistema. Por ello, se recomienda el uso de herramientas como Snyk, WhiteSource o Dependabot, que escanean automáticamente las dependencias y alertan sobre vulnerabilidades CVE registradas (Sonatype, 2023).

Estas herramientas permiten establecer políticas de bloqueo de versiones inseguras, automatizar actualizaciones y aplicar parches, así como generar reportes de cumplimiento de seguridad como parte del proceso de entrega continua.

4.11. Evaluación de Impacto de la Migración Arquitectónica

La migración de una arquitectura monolítica hacia una basada en microservicios implica cambios significativos tanto en el plano técnico como organizacional. Para evaluar el éxito de esta transformación, es fundamental establecer un conjunto de indicadores cuantificables que permitan comparar el comportamiento del sistema antes y después de la implementación. Asimismo, es necesario identificar los beneficios tangibles e intangibles que esta evolución aporta, en términos de calidad del software, eficiencia operativa y retorno sobre la inversión (Newman S. , 2019).

4.11.1. Indicadores de calidad: rendimiento, tiempo de respuesta, MTTR

Los indicadores de calidad del sistema permiten medir objetivamente el impacto de una decisión arquitectónica. En el caso de los microservicios, algunos de los más relevantes son el rendimiento (capacidad del sistema para procesar solicitudes bajo carga), el tiempo de respuesta (latencia promedio entre la solicitud y la respuesta), y el MTTR (Mean Time to Recovery), que mide el tiempo medio necesario para restaurar un servicio tras una falla (Humble, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 2010).

La arquitectura de microservicios, al estar compuesta por componentes desacoplados y autónomos, mejora estos indicadores gracias a la escalabilidad horizontal, el despliegue independiente de servicios y la tolerancia a fallos localizados (Newman S. , 2019). Además, herramientas modernas como Prometheus, Grafana y Jaeger permiten recolectar métricas detalladas para construir una línea base y monitorear la evolución post-migración (Galin, 2017)

4.11.2. Cuadro comparativo antes vs. después (benchmark técnico)

Para evidenciar los beneficios de la migración, se recomienda realizar benchmarks técnicos que comparen el desempeño de la arquitectura monolítica versus la de

microservicios. Esta comparación puede realizarse a través de pruebas de carga, simulaciones o pilotos controlados.

Un ejemplo de análisis comparativo podría verse así:

Indicador	Arquitectura Monolítica	Arquitectura de Microservicios	Mejora estimada (%)
Tiempo promedio de respuesta	2.5 s	0.9 s	64%
MTTR	4 horas	45 minutos	81%
Escalabilidad	Limitada	Horizontal y granular	Alta
Tiempo promedio de despliegue	60 minutos	5 minutos	91%

Tabla 2. Ejemplo de análisis de arquitecturas. Realizado por: Ing. Marco Clavijo - Ing. Amanda Cabascango

Esta información permite comunicar de forma clara el valor agregado de la migración tanto al área técnica como a los tomadores de decisiones (Galin, 2017).

4.11.3. Modelos de costo-beneficio, ROI técnico y organizacional

La toma de decisiones sobre la evolución arquitectónica debe apoyarse en un análisis de costo-beneficio que contemple tanto factores técnicos como estratégicos. Este análisis debe considerar:

- Costos directos: licencias, horas de desarrollo, infraestructura, capacitación, herramientas.
- Costos indirectos: curva de aprendizaje, resistencia al cambio, riesgos operativos.
- Beneficios técnicos: menor tiempo de respuesta, menor MTTR, mayor disponibilidad.
- Beneficios organizacionales: entregas más rápidas, mejor experiencia del usuario, autonomía de equipos

Por ejemplo, si la organización evita tres horas de caída mensual a un costo estimado de $2,000 por hora, el ahorro anual sería de $72,000. Si el proyecto de migración costó $45,000, el ROI sería del 60% en un año (Galin, 2017).

Como recomienda (Newman S. , 2019), el análisis de retorno no debe limitarse al ahorro económico inmediato, sino también considerar el valor técnico y organizacional generado: sostenibilidad del software, reducción de riesgos, alineación con objetivos estratégicos y mejor capacidad de respuesta frente a incidentes.

4.11.4. Contextos delimitados (Bounded Contexts)

El concepto de Bounded Contexts proviene del enfoque de Domain-Driven Design y se refiere a la delimitación explícita de un dominio en subdominios claramente separados, cada uno con su propio modelo, lenguaje y reglas (Evans, 2003) . Esta separación permite evitar ambigüedades semánticas y reducir la complejidad interna del sistema, ya que cada contexto opera de manera autónoma y se integra con otros mediante interfaces o contratos bien definidos. En arquitecturas basadas en microservicios, los Bounded Contexts son fundamentales para garantizar que cada servicio mantenga coherencia funcional y una responsabilidad específica. (Fowler M. , Bounded Context, 2014)

5. Materiales y metodología

5.1. Introducción

El desarrollo de este proyecto se centró en el diseño de una arquitectura de software basada en microservicios, orientada a las necesidades de una empresa dedicada a la venta y distribución de servicios de energía eléctrica. Por lo cual se adoptó la metodología ágil Scrum, debido a su enfoque iterativo, adaptable y colaborativo.

Aunque el proyecto no incluye una implementación, se aplicaron los principios de Scrum para organizar las actividades en sprints conceptuales, realizar entregables incrementales (documentación arquitectónica y diagramas), y facilitar reuniones periódicas con stakeholders para validar el avance y ajustar el diseño.

5.2. Análisis de requisitos

El diseño de la arquitectura propuesta se fundamenta en un análisis estructurado de los requisitos funcionales y no funcionales del sistema, considerando tanto las necesidades operativas actuales del SDI como las restricciones tecnológicas de la empresa eléctrica.

5.2.1. Requisitos funcionales

Se identificaron los requisitos funcionales vinculados a los procesos de negocio: registro y seguimiento de reclamos, generación y cierre de reparaciones, gestión de grupos de trabajo, consulta de catálogos operativos y envío de notificaciones. Estos procesos demandan una arquitectura capaz de representar correctamente los flujos transaccionales, mantener la trazabilidad de las operaciones y soportar integraciones con sistemas ya existentes, como RecepcionesWS y servicios de correo. La arquitectura debía, además, permitir la evolución independiente de cada dominio para introducir nuevas funcionalidades sin afectar al resto del ecosistema.

5.2.2. Requisitos no funcionales

Los requisitos no funcionales, el análisis reveló necesidades claras en cuatro áreas críticas: escalabilidad, resiliencia, seguridad y observabilidad. Desde la perspectiva de escalabilidad, el sistema debía soportar incrementos en la carga del Call Center y responder adecuadamente ante picos derivados de eventos climáticos o campañas operativas. Esto requirió diseñar una arquitectura capaz de escalar horizontalmente, especialmente en los dominios de reclamos y reparaciones. En términos de resiliencia, se estableció como requisito la continuidad del servicio incluso ante fallas parciales de componentes o servicios externos, lo que justificó el uso de patrones como circuit breakers, colas, reintentos y Sagas.

Respecto a la seguridad, la arquitectura debía alinearse a los estándares corporativos, incorporando autenticación centralizada (OIDC), autorización basada en roles y atributos, protección de APIs mediante Gateway y cifrado en tránsito y en reposo. Finalmente, se identificó la necesidad de garantizar observabilidad integral del sistema mediante métricas, logs estructurados y trazas distribuidas que permitan diagnosticar problemas de forma proactiva y mejorar la operación continua.

En conjunto, estos requisitos guiaron el diseño de una arquitectura modular, desacoplada y orientada a eventos, apta para evolucionar a largo plazo y alineada con las necesidades estratégicas de la empresa eléctrica.

5.3. Metodología de Trabajo

5.3.1. Enfoque metodológico

El enfoque metodológico adoptado para esta investigación es de tipo aplicado, con una orientación descriptiva–analítica y basada en principios de arquitectura de software y buenas prácticas de ingeniería. El objetivo central es proponer una arquitectura de microservicios adecuada para el Sistema de Distribución de Información (SDI) en los módulos de reparaciones y reclamos, tomando como punto

de partida el análisis de la situación actual, las necesidades de la organización y los lineamientos modernos de diseño de sistemas distribuidos.

Este enfoque se estructura en cuatro pilares:

1. Análisis estructurado del estado actual

Se examinan los componentes existentes, su arquitectura, sus limitaciones técnicas y operativas, y la forma en que interactúan los usuarios, los módulos funcionales y los sistemas externos. Este análisis permite identificar las brechas, cuellos de botella y riesgos asociados a la solución monolítica vigente.

2. Aplicación de principios de arquitectura moderna

Se emplean lineamientos reconocidos en la industria —como microservicios, separación de dominios, patrones de integración, prácticas de observabilidad y seguridad— con el fin de diseñar una propuesta arquitectónica coherente, escalable y alineada con las necesidades de una empresa eléctrica. La metodología se fundamenta en marcos como Domain-Driven Design (DDD), patrones de integración (API Gateway, ACL, mensajería), patrones de resiliencia

3. Modelado conceptual y arquitectónico

Se elaboran modelos y diagramas (C4 Model: contexto, contenedores y componentes) que permiten representar la estructura de la arquitectura propuesta sin necesidad de llegar al nivel de implementación. Estos modelos facilitan la comunicación técnica y la comprensión del nuevo diseño por parte de stakeholders, arquitectos y equipos de desarrollo.

4. Evaluación de viabilidad y alineación con la organización

El enfoque metodológico considera aspectos operativos, tecnológicos y organizacionales de la empresa, evaluando la factibilidad del diseño, los beneficios esperados, los riesgos potenciales y las condiciones mínimas necesarias para su

adopción. Este proceso asegura que la arquitectura propuesta sea pertinente y aplicable en el contexto real.

La adopción de Scrum se reflejó en los siguientes componentes:

Sprints: Se definieron sprints de 1 semana, con objetivos específicos como levantamiento de requerimientos, definición de microservicios, modelado de diagramas, entre otros.

Eventos:

- **Sprint Planning:** Para definir los entregables de cada iteración.
- **Daily Scrum**: No se aplicó diariamente por la motivos laborales y académicos, pero se hicieron reuniones de control semanales.
- **Sprint Review:** Se compartieron los avances con stakeholders al final de cada sprint conceptual.
- **Sprint Retrospective:** Se revisó el enfoque seguido para introducir mejoras en el siguiente ciclo.

Roles y artefactos utilizados

Para el desarrollo del proyecto se establecieron roles claros dentro del marco de trabajo Scrum:

- Product Owner (Jefe del Departamento de Desarrollo): Representa las necesidades del negocio y se encarga de priorizar los requerimientos según su impacto y valor.
- Scrum Master (Tutor): Facilita el proceso de trabajo, guía la aplicación correcta de la metodología y ayuda a resolver impedimentos.
- Equipo de Desarrollo: Conformado por quienes realizan el diseño técnico y conceptual del sistema, incluyendo la elaboración de modelos, arquitectura y componentes principales.

Actividades realizadas por sprint

Sprint 1: Levantamiento y comprensión del estado actual

- Revisión de la documentación existente de los módulos de reclamos y reparaciones.
- Análisis del funcionamiento operativo del sistema SDI y sus dependencias.
- Identificación de actores, procesos y flujos de información principales.
- Inventario de tecnologías, herramientas y patrones utilizados en el sistema actual.
- Detección inicial de problemas, limitaciones técnicas y oportunidades de mejora.
- Elaboración del mapa general del sistema monolítico actual.

Sprint 2: Análisis de dominios y definición de bounded contexts

- Descomposición funcional de los módulos actuales en dominios de negocio.
- Identificación de entidades, procesos clave y reglas relevantes para el diseño.
- Delimitación de bounded contexts siguiendo principios de Domain-Driven Design (DDD).
- Propuesta preliminar de los microservicios candidatos según responsabilidad.
- Validación conceptual de los bounded contexts frente a los requerimientos del SDI.
- Documentación del modelo conceptual resultante.

Sprint 3: Diseño arquitectónico de alto nivel

- Selección y justificación del estilo arquitectónico basado en microservicios.
- Definición del diagrama de contexto del sistema propuesto (Modelo C4 – Nivel 1).
- Identificación de los contenedores (microservicios, bases de datos, broker, API Gateway).
- Evaluación de patrones de comunicación sincrónica y asincrónica.

- Definición inicial de patrones transversales: seguridad, observabilidad y resiliencia.
- Documentación de la arquitectura general en su versión preliminar.

Sprint 4: Modelado detallado de la solución y validación técnica

- Elaboración del diagrama de contenedores (Modelo C4 – Nivel 3).
- Definición de responsabilidades internas de cada microservicio y sus interacciones.
- Identificación de los flujos críticos (registro de reclamos, generación de reparaciones, asignación de grupos de trabajo).
- Evaluación técnica de tecnologías posibles (API Gateway, mensajería, bases de datos).
- Análisis de riesgos, supuestos y dependencias técnicas del diseño.
- Ajustes a la propuesta arquitectónica tras el análisis de viabilidad.

Sprint 5: Refinamiento, documentación final y elaboración del plan de migración

- Consolidación de toda la documentación técnica elaborada en sprints anteriores.
- Integración del plan de migración gradual basado en el patrón Strangler Fig.
- Redacción del capítulo de conclusiones y recomendaciones arquitectónicas.
- Validación final de coherencia entre problemas identificados y solución propuesta.
- Preparación de anexos, diagramas y estructura final de la tesis

5.4. Diseño de la arquitectura (Nivel Arquitectónico)

En esta sección se presenta el diseño arquitectónico propuesto para la Empresa Eléctrica, orientado a modernizar la gestión de reclamos y reparaciones mediante una arquitectura basada en microservicios. Se describen los dominios clave, los componentes arquitectónicos y la forma en que estos interactúan para mejorar eficiencia, escalabilidad y continuidad operativa. Asimismo, se emplea el modelo C4

para representar la solución en niveles de contexto y contenedores, facilitando una visión clara y estructurada de la propuesta.

5.4.1. Principios de diseño aplicados

El diseño arquitectónico propuesto se fundamenta en un conjunto de principios que garantizan una solución moderna, escalable y alineada a las necesidades operativas de la Empresa Eléctrica. En primer lugar, se adopta el principio de separación de responsabilidades, lo que permite dividir el dominio de reclamos y reparaciones en unidades independientes y coherentes. Asimismo, se aplica el diseño orientado a dominios (DDD) para asegurar que cada componente responda fielmente a los procesos y reglas del negocio eléctrico.

Se incorpora además el principio de desacoplamiento, promoviendo que los servicios interactúen mediante contratos bien definidos y reduciendo dependencias rígidas. Complementariamente, se considera la observabilidad como pilar transversal, incluyendo prácticas de trazabilidad, métricas y monitoreo que faciliten la continuidad operativa y la detección temprana de fallas.

Finalmente, se aplican principios de resiliencia, seguridad y escalabilidad, esenciales para una infraestructura crítica como la del sector eléctrico. Estos principios garantizan que la arquitectura propuesta sea robusta, flexible ante cambios y capaz de soportar el crecimiento futuro de la organización.

5.4.2. Identificación de contextos delimitados

La identificación de Contextos delimitados constituye un paso esencial para definir los límites funcionales y semánticos de la arquitectura propuesta. A partir del análisis del modelo operativo de la Empresa Eléctrica, se determinaron los dominios de negocio que intervienen directamente en la gestión de reclamos y reparaciones, permitiendo delimitar áreas independientes con reglas, datos y procesos propios. Esta separación facilita la organización del sistema en módulos autónomos, coherentes y fácilmente escalables.

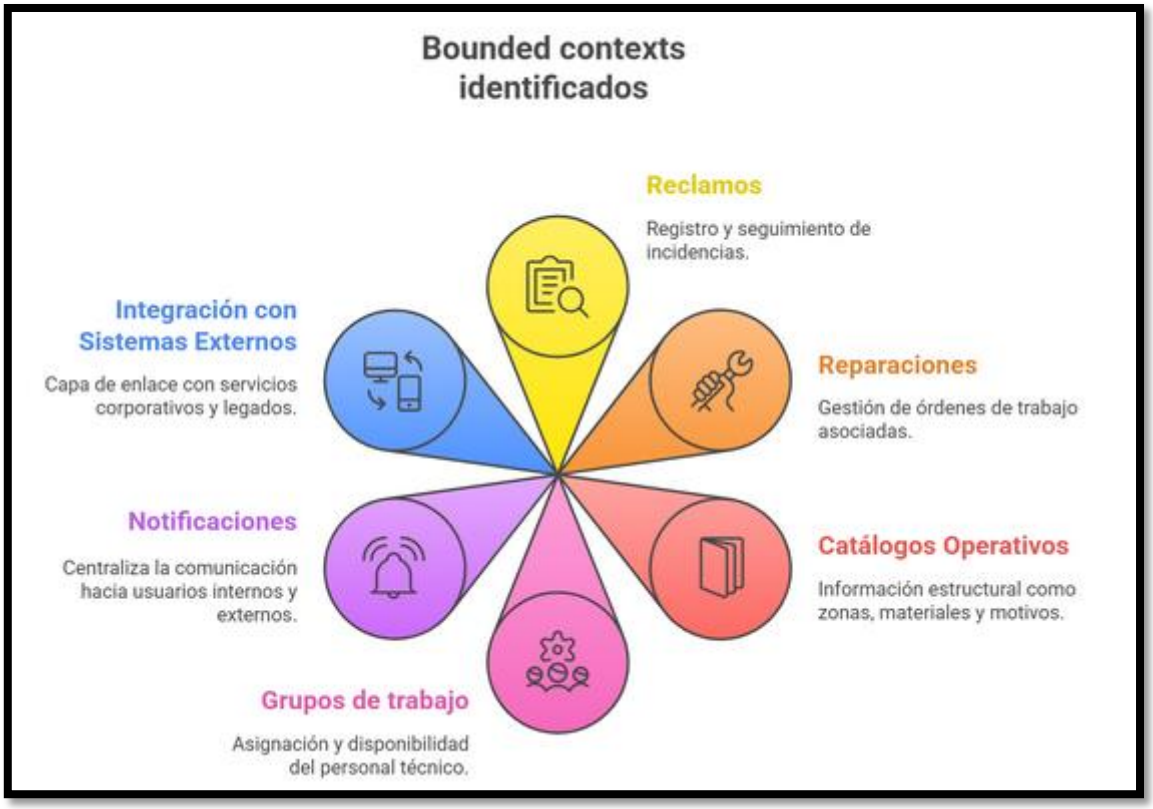

Figura 2. Boundes Contexts Identificados. Realizado por: Ing. Marco Clavijo. Imagen creada con IA: https://www.napkin.ai

Cada contexto fue delimitado considerando sus actores, procesos, fuentes de datos y reglas de negocio, asegurando que posea un propósito definido y no invada las responsabilidades de otro dominio. Esta estructuración permite que la propuesta arquitectónica refleje fielmente la operación de la Empresa Eléctrica, favoreciendo el desacoplamiento, la claridad funcional y la evolución futura de cada componente.

5.4.3. Definición de capas arquitectónicas

La arquitectura propuesta se organiza en un conjunto de capas que permiten estructurar de forma ordenada las responsabilidades del sistema, garantizando claridad, mantenibilidad y alineación con las necesidades operativas de la Empresa Eléctrica. Esta división facilita la escalabilidad, la resiliencia y la correcta evolución de los procesos asociados a la gestión de reclamos y reparaciones.

Capa de Experiencia reúne los puntos de interacción con los usuarios, como operadores del Call Center, personal técnico y áreas administrativas. Su objetivo es

proporcionar una experiencia unificada y simplificada, ocultando la complejidad interna de los servicios y dominios que componen la arquitectura.

Capa de Acceso o API funciona como la puerta de entrada hacia la solución. A través del uso de un API Gateway o mecanismos de control corporativo, esta capa centraliza la autenticación, autorización, enrutamiento, políticas de seguridad y control de tráfico. Además, garantiza la interoperabilidad con los lineamientos tecnológicos de la organización.

Capa de Negocio se encarga de la orquestación de los procesos transversales y de la coordinación entre los diferentes dominios. Aquí residen los servicios de aplicación responsables de ejecutar casos de uso completos, integrar múltiples microservicios, aplicar validaciones compuestas y manejar flujos como "registro de reclamo – generación de reparación – asignación de grupos de trabajo". Esta capa evita que los microservicios del dominio se sobrecarguen con lógica de coordinación y asegura una ejecución coherente de los procesos.

Capa de Dominio representa el núcleo funcional de la propuesta. En ella se encuentran los microservicios correspondientes a los bounded contexts identificados (Reclamos, Reparaciones, Catálogos, Grupos de Trabajo, Notificaciones e Integraciones). Cada uno encapsula su propia lógica de negocio, reglas, datos y eventos, proporcionando independencia y facilitando la evolución aislada de cada área del negocio eléctrico.

Finalmente, la **Capa de Datos** agrupa los repositorios de información de cada dominio, administrados preferiblemente bajo el principio de "base de datos por servicio" o, en su fase transitoria, mediante esquemas dedicados. Esta capa garantiza integridad, disponibilidad y consistencia de los datos que respaldan las operaciones críticas de la Empresa Eléctrica.

La definición estructurada de estas capas consolida una arquitectura modular, escalable y preparada para un entorno de misión crítica, permitiendo que las áreas operativas de la Empresa Eléctrica cuenten con una solución moderna, robusta y alineada a sus exigencias futuras.

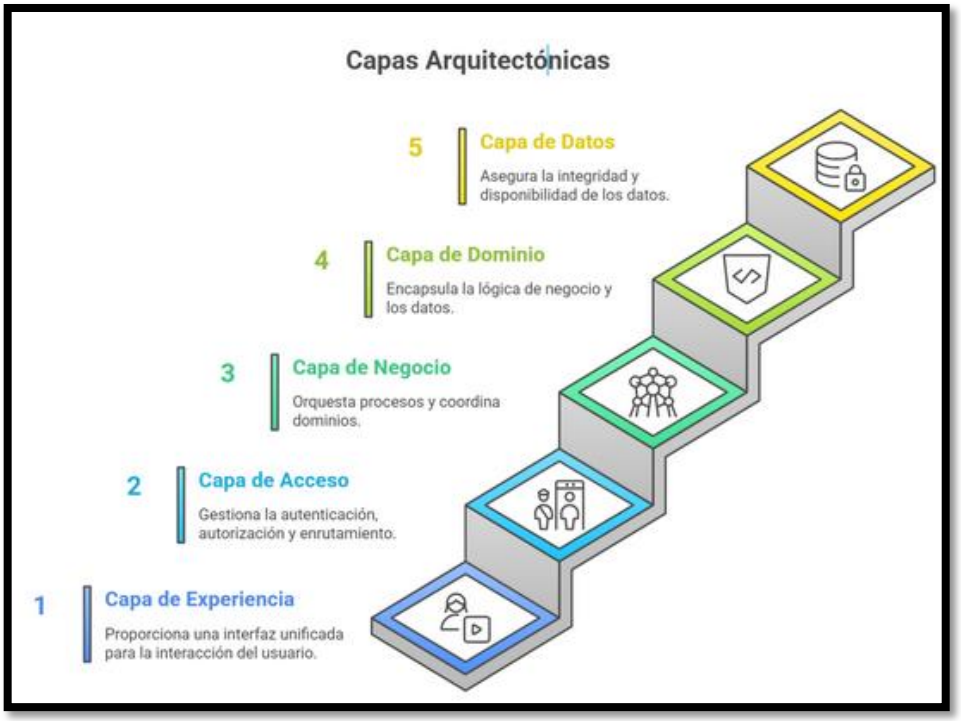

Figura 3. Descripción de capas arquitectónicas de la arquitectura propuesta. Realizado por: Ing. Marco Clavijo. Imagen creada con IA: https://www.napkin.ai

5.4.4. Tipos de comunicación y resiliencia

La arquitectura propuesta combina dos tipos de comunicación:

- REST (sincrónica) para consultas rápidas o acciones que requieren respuesta inmediata, como obtener datos de un reclamo o verificar información de un catálogo.
- Mensajería (asincrónica) para procesos que pueden ejecutarse en segundo plano, como generar una reparación, enviar notificaciones o asignar grupos de trabajo. Esto evita bloqueos y permite que cada servicio siga trabajando aunque otro esté temporalmente ocupado.

Para asegurar la resiliencia, se aplican mecanismos sencillos pero fundamentales: timeouts para evitar esperas indefinidas, reintentos automáticos cuando un servicio falla momentáneamente, y circuit breakers para evitar que un fallo en un servicio afecte a los demás. Además, se garantiza que las operaciones importantes sean idempotentes, de modo que si se repiten no generen duplicados ni inconsistencias.

Con estos patrones, la arquitectura se vuelve más estable, tolerante a fallas y capaz de seguir funcionando incluso en momentos de alta demanda, asegurando la continuidad operativa en los procesos de reclamos y reparaciones de la Empresa Eléctrica.

6. Desarrollo del Proyecto (Análisis y Propuesta Arquitectónica)

6.1. Análisis de arquitectura actual

En esta sección se examinan los sistemas actuales utilizados por la Empresa Eléctrica para la gestión de reclamos y reparaciones, evaluando su arquitectura, funcionamiento y componentes principales. Se identifican las limitaciones técnicas del modelo monolítico, así como los problemas de escalabilidad, mantenimiento y dependencia entre módulos. Además, se analizan los flujos de información, los procesos operativos y las integraciones externas que soportan las operaciones diarias. Este análisis permite establecer una línea base clara para justificar la necesidad de una arquitectura moderna y más flexible.

6.2. Modernización de Aplicaciones Legacy: Caso Oracle Forms

La modernización de aplicaciones legacy es un componente estratégico en la transformación digital de organizaciones que dependen de sistemas tradicionales para gestionar procesos críticos[1]. En particular, tecnologías como Oracle Forms, ampliamente utilizadas en sectores como energía, banca y administración pública, requieren enfoques específicos para evolucionar hacia arquitecturas modernas como los microservicios.

6.2.1. Qué es Oracle Forms y su rol en aplicaciones empresariales

Oracle Forms es una herramienta de desarrollo declarativo creada por Oracle Corporation que permite construir interfaces gráficas conectadas directamente con bases de datos Oracle. Fue concebida para facilitar el desarrollo de aplicaciones

empresariales transaccionales, sobre todo en arquitecturas cliente-servidor, y más tarde en modelos web (Deshpande, 2016).

Esta plataforma fue ampliamente utilizada en la creación de sistemas de misión crítica como facturación, inventario y recursos humanos, debido a su integración nativa con Oracle Database y a su bajo requerimiento de codificación compleja. Su facilidad para construir formularios, validaciones y operaciones sobre datos relacionales convirtió a Oracle Forms en una solución dominante en muchas empresas desde los años 90 hasta la actualidad (Corporation, 2022).

No obstante, a pesar de su estabilidad y rendimiento, Oracle Forms no responde a las exigencias de flexibilidad, interoperabilidad y escalabilidad que requieren los entornos actuales, sobre todo aquellos que se orientan hacia arquitecturas distribuidas basadas en microservicios.

6.2.2. Limitaciones frente a arquitecturas modernas

Las aplicaciones basadas en Oracle Forms presentan una serie de limitaciones estructurales y tecnológicas frente a los enfoques modernos de desarrollo de software. En primer lugar, su naturaleza monolítica genera acoplamientos fuertes entre la lógica de presentación, negocio y acceso a datos, dificultando la escalabilidad horizontal y la implementación de cambios aislados (Newman, 2019).

Adicionalmente, muchas implementaciones dependen de tecnologías desactualizadas como Applets Java o Java Web Start, que han perdido soporte en los navegadores actuales, lo cual compromete la accesibilidad y seguridad de los sistemas existentes (Corporation, 2022).

Otro punto crítico es la incompatibilidad con arquitecturas orientadas a servicios (SOA o microservicios). Oracle Forms no fue diseñado para exponer ni consumir APIs REST, dificultando la integración con otros sistemas o con plataformas en la nube. Además, sus interfaces gráficas no son responsivas ni adaptables a dispositivos móviles, lo que limita la experiencia del usuario final (Deshpande A. , 2016).

6.2.3. Arquitectura actual de moduloReparaciones

Figura 4. Arquitectura Actual del departamento. Realizado por: Ing. Amanda Cabascango

El módulo moduloReparaciones opera bajo una arquitectura monolítica desarrollada en Java EE 8, desplegada como un archivo WAR sobre un servidor de aplicaciones Oracle. Su estructura combina en un mismo componente la interfaz de usuario (basada en JSF y PrimeFaces), la lógica de negocio y el acceso directo a la base de datos mediante JDBC. La mayor parte del procesamiento se realiza desde controladores vinculados a las pantallas, lo que genera un fuerte acoplamiento entre la lógica operativa y la capa visual.

El acceso a datos se ejecuta a través de clases DAO con consultas SQL embebidas, sin uso real de JPA y sin un mecanismo de administración de conexiones optimizado, lo que incrementa la posibilidad de cuellos de botella. Asimismo, el módulo mantiene estado en sesión y depende de múltiples catálogos y entidades internas para procesar reparaciones, recepciones y asignaciones. Esta arquitectura limita la escalabilidad, dificulta el mantenimiento y reduce la flexibilidad para introducir nuevas funcionalidades o integraciones.

6.2.4. Arquitectura actual de sdi_reclamos

El módulo sdi_reclamos está construido bajo una arquitectura monolítica similar a la de moduloReparaciones, utilizando Java EE 8 y desplegándose como un archivo WAR sobre un servidor de aplicaciones Oracle. Su diseño integra en un mismo

componente la interfaz de usuario desarrollada con JSF y PrimeFaces, la lógica de negocio implementada directamente en los controladores, y el acceso a la base de datos mediante DAOs que utilizan JDBC puro. Esta estructura genera un fuerte acoplamiento entre las capas, dificultando la separación de responsabilidades y la evolución modular del sistema.

El módulo administra procesos relacionados con el registro, clasificación y seguimiento de reclamos tanto de Operación y Mantenimiento (OM) como de Alumbrado Público (AP). Para ello, utiliza controladores y páginas JSF que dependen de entidades y catálogos internos. La gestión de datos se realiza mediante consultas SQL embebidas dentro de los DAOs, sin aprovechamiento de JPA ni mecanismos avanzados de persistencia, lo que incrementa la carga operacional y la posibilidad de inconsistencias.

Adicionalmente, sdi_reclamos mantiene estado en la sesión del usuario y utiliza parámetros definidos en archivos de configuración para integrarse con servicios externos, como sistemas de correo o herramientas de Call Center. Esta arquitectura, aunque operativa, presenta limitaciones en escalabilidad, mantenibilidad, seguridad y capacidad de integración, especialmente frente a las necesidades actuales de modernización y operación continua de la Empresa Eléctrica.

6.2.5. Problemas y deuda técnica identificada

El análisis de los sistemas actuales evidenció un conjunto de problemas técnicos significativos y una acumulación considerable de deuda tecnológica que limita la evolución, mantenibilidad y estabilidad de los módulos moduloReparaciones y sdi_reclamos. Ambos sistemas comparten una arquitectura monolítica.

A) Arquitectura monolítica altamente acoplada

Ambos módulos están implementados como aplicaciones WAR en Java EE 8, combinando interfaz, lógica de negocio y acceso a datos en un único artefacto. Esto genera un fuerte acoplamiento que dificulta la evolución independiente de

funcionalidades, incrementa el riesgo ante cambios y limita la modularidad necesaria para un entorno crítico como el de la Empresa Eléctrica.

B) Lógica de negocio mezclada con la capa de presentación

Se observa que los controladores JSF contienen reglas de negocio, validaciones y lógica operativa, rompiendo el principio de separación de responsabilidades. Este diseño complica las pruebas unitarias, favorece la duplicación de código y dificulta mantener un flujo de negocio coherente.

C) Acceso a datos basado en JDBC con consultas embebidas

El uso intensivo de DAOs con SQL incrustado genera fragilidad, reduce la portabilidad y hace compleja la evolución de la capa de persistencia. La falta de una capa de abstracción genera alto acoplamiento entre el modelo lógico y el modelo físico de la base de datos.

D) Ausencia de ORM y falta de modelado de entidades

El archivo persistence.xml no configura un ORM ni entidades JPA, lo que evidencia una ausencia total de modelado de objetos persistentes. Esto aumenta la probabilidad de inconsistencias, requiere manejo manual de transacciones y complica auditorías, migraciones y mantenibilidad a largo plazo.

E) Falta de un pool de conexiones eficiente

El acceso a base de datos utiliza conexiones JDBC manuales sin un administrador de conexiones moderno (como HikariCP). Esto incrementa el riesgo de agotamiento de conexiones, especialmente en momentos de alta demanda, afectando directamente la disponibilidad del sistema.

F) Integraciones externas sin resiliencia

Los módulos realizan llamadas directas a servicios externos (por ejemplo, RecepcionesWS y correo) sin aplicar mecanismos como timeouts, reintentos, circuit

breakers o colas. Esta ausencia de resiliencia puede provocar bloqueos, fallas en cascada y pérdida de información cuando los sistemas externos no responden oportunamente.

G) Ausencia de una capa de integración (ACL)

No existe un Anti-Corruption Layer que proteja a los módulos actuales de cambios en sistemas corporativos externos. Esto hace que cualquier modificación en los servicios externos pueda romper funcionalidades internas, incrementando la fragilidad del sistema.

H) Falta de observabilidad moderna

Los sistemas carecen de logs estructurados, métricas y trazas distribuidas. Esto dificulta el diagnóstico de fallas, la optimización del rendimiento y la aplicación de monitoreo proactivo, afectando la capacidad de soportar operaciones 24/7.

I) Manejo de estado en servidor

El uso de HttpSession para almacenar información del usuario limita la capacidad de escalar horizontalmente, ya que la sesión queda dependiente de la instancia del servidor, impidiendo un balanceo de carga eficiente.

J) Prácticas débiles de seguridad

Se detecta manejo de credenciales en texto plano, configuración embebida en archivos internos del WAR y ausencia de estándares modernos como OIDC, JWT o gestión centralizada de secretos. Esto representa riesgos importantes desde el punto de vista de seguridad corporativa.

K) Dependencia de catálogos y configuraciones internas

Ambos módulos acceden a catálogos operativos y parámetros desde el monolito sin mecanismos de cache, versionamiento o desacoplamiento, lo que provoca duplicación de código, riesgo de inconsistencias y un alto costo de mantenimiento.

L) Escasa escalabilidad y poca flexibilidad operativa

La combinación de estado en sesión, JDBC manual, arquitectura monolítica y falta de resiliencia provoca que los sistemas no puedan escalar adecuadamente ante aumentos de demanda, ni adaptarse fácilmente a nuevas funcionalidades o integraciones.

6.2.6. Análisis de Restricciones y Limitaciones

En esta sección se identifican las principales restricciones y limitaciones que afectan a los módulos actuales de la Empresa Eléctrica en la gestión de reclamos y reparaciones. Entre ellas destacan las limitaciones tecnológicas derivadas del uso de una arquitectura monolítica, la dependencia de infraestructura legada, las restricciones de escalabilidad por manejo de sesiones y el acceso a datos mediante JDBC. Asimismo, se consideran limitaciones operativas relacionadas con integraciones externas, falta de observabilidad y ausencia de mecanismos modernos de seguridad. Estas restricciones condicionan la capacidad de evolución del sistema y justifican la necesidad de adoptar una arquitectura más flexible y moderna.

6.2.7. Restricciones tecnológicas

Las soluciones actuales presentan diversas restricciones tecnológicas que limitan su rendimiento, escalabilidad y capacidad de evolución. En primer lugar, ambos módulos operan sobre una arquitectura monolítica basada en Java EE 8, tecnología que ha quedado desactualizada frente a los estándares modernos de desarrollo distribuido. Esta plataforma restringe la adopción de prácticas como contenedorización ligera, escalado independiente y despliegues ágiles.

Asimismo, el acceso a datos mediante JDBC puro y la inexistencia de un ORM dificultan la optimización, mantenimiento y portabilidad de la información. La falta de observabilidad nativa, la ausencia de mecanismos de resiliencia y los escasos controles de seguridad generan además limitaciones importantes para operar en entornos de alta disponibilidad. Finalmente, la dependencia de infraestructura

legada y de integraciones externas poco flexibles condiciona la capacidad de transición hacia arquitecturas más modernas, robustas y modulares.

6.2.8. Restricciones operativas

Las principales restricciones operativas identificadas son:

- Necesidad de operación continua (24/7): Los módulos no pueden detenerse por largos periodos debido a la naturaleza crítica del servicio eléctrico.
- Dependencia de múltiples áreas internas: Call Center, grupos de trabajo, supervisores y personal técnico dependen directamente del sistema, dificultando cambios disruptivos.
- Limitada capacidad para enfrentar picos de demanda: Eventos como fallas masivas incrementan la carga y exponen limitaciones de escalabilidad.
- Dificultad para realizar mantenimientos programados: La operación diaria impide ventanas amplias para aplicar actualizaciones o refactorizaciones.
- Integraciones rígidas con sistemas corporativos: Cambios en los servicios externos no pueden propagarse rápidamente, restringiendo la adaptabilidad.
- Ausencia de monitoreo operativo: No existen métricas ni alertas que permitan anticipar incidentes o degradación del servicio.
- Dependencia de flujos manuales o semi-automatizados: Algunas tareas requieren intervención humana, aumentando el riesgo operativo y dificultando optimizaciones.

6.2.9. Matriz de Probabilidad vs Impacto

Riesgo identificado	Probabilidad	Impacto	Nivel de Riesgo
Saturación o agotamiento de conexiones JDBC	Alta	Alta	**Crítico**
Caída total del monolito por falta de aislamiento de fallas	Media	Alta	**Alto**
Fallas en integraciones externas sin timeouts/circuit breakers	Alta	Media	**Alto**
Lentitud o indisponibilidad por ausencia de escalabilidad	Alta	Alta	**Crítico**

Falta de monitoreo que retrasa la detección de incidentes	Media	Media	**Moderado**
Pérdida o inconsistencia de datos por ausencia de ORM y transacciones estructuradas	Baja	Alta	**Alto**
Dependencia de sesiones en servidor que impide escalar horizontalmente	Media	Media	**Moderado**
Caídas por picos de demanda en eventos masivos (fallas eléctricas)	Alta	Alta	**Crítico**
Vulnerabilidades por manejo de credenciales en texto plano	Baja	Alta	**Alto**
Indisponibilidad durante mantenimientos debido a arquitectura rígida	Media	Media	**Moderado**

Tabla 3. Matriz comparativa de arquitectura actual vs. Propuesta. Realizado por: Ing. Marco Clavijo - Ing. Amanda Cabascango

6.3. Diseño Arquitectónico Propuesto

La propuesta plantea una arquitectura basada en microservicios que divide el sistema en componentes independientes y fáciles de mantener. Se organiza en capas bien definidas y combina comunicación REST y mensajería para mejorar resiliencia y rendimiento. Con ello, la Empresa Eléctrica obtiene una solución más flexible, escalable y preparada para futuras necesidades.

6.3.1. Solución arquitectónica

A) Objetivos de la arquitectura

La arquitectura propuesta tiene como objetivo principal modernizar la gestión de reclamos y reparaciones mediante una estructura más flexible, modular y fácil de mantener. Busca mejorar la escalabilidad del sistema, permitir la evolución independiente de cada componente y asegurar una operación continua incluso ante picos de demanda o fallas en servicios externos. Además, pretende fortalecer la seguridad, facilitar la integración con sistemas corporativos y garantizar una mayor eficiencia en los procesos operativos de la Empresa Eléctrica.

B) Microservicios por dominio

La arquitectura propuesta organiza la solución en microservicios definidos a partir de los dominios clave de la Empresa Eléctrica, permitiendo que cada uno gestione sus propios datos, procesos y reglas de negocio. Entre los dominios identificados se encuentran: Reclamos, encargado del registro y seguimiento de incidencias;

Reparaciones, orientado a la generación y cierre de órdenes de trabajo; Catálogos Operativos, que centraliza información estructural como motivos, zonas y materiales; Grupo de Trabajos, responsable de la asignación y disponibilidad del personal técnico; Notificaciones, dedicado al envío de avisos internos y externos; y Integración Externa, que actúa como puente hacia sistemas corporativos.

Cada microservicio opera de forma independiente, exponiendo APIs específicas y comunicándose mediante eventos cuando es necesario coordinar procesos entre dominios. Esta organización facilita la escalabilidad, reduce el acoplamiento y permite que cada área evolucione sin afectar a las demás, fortaleciendo así la capacidad operativa de la Empresa Eléctrica.

C) Capa de Negocio

La Capa de Negocio constituye el nivel intermedio entre la capa de acceso y los microservicios de dominio, y tiene como propósito principal coordinar y orquestar los procesos funcionales que involucran a múltiples dominios dentro de la Empresa Eléctrica. Su rol es fundamental para garantizar coherencia, trazabilidad y consistencia en los flujos operativos que no pueden ser manejados de forma aislada por un solo microservicio.

En esta capa se implementan servicios de aplicación encargados de ejecutar casos de uso completos, tales como: registrar un reclamo, generar la reparación correspondiente, validar la disponibilidad de grupos de trabajo, asignarlas y notificar a los actores involucrados. Cada uno de estos pasos podría pertenecer a dominios distintos; por ello, la Capa de Negocio actúa como un coordinador que encadena las acciones necesarias, evitando que la lógica de flujo quede distribuida o duplicada dentro de los microservicios.

Además, esta capa incorpora mecanismos de orquestación avanzada, tales como Sagas para manejar transacciones distribuidas, validaciones compuestas que afectan a varios agregados, y políticas de compensación para revertir pasos cuando un proceso falla en un punto intermedio. De esta manera se garantiza la integridad

del flujo operativo y se reduce la dependencia de llamadas sincrónicas entre servicios.

Otro aspecto clave es que la Capa de Negocio centraliza reglas transversales que aplican a más de un dominio, como políticas de auditoría, consistencia de datos, condiciones operativas eléctricas y reglas que afectan simultáneamente a reclamos, reparaciones y grupos de trabajo. Esto evita duplicidad de lógica, mejora la mantenibilidad y facilita la incorporación de nuevas funcionalidades sin afectar el funcionamiento interno de cada microservicio.

En conjunto, esta capa aporta orden, control y coherencia a la solución arquitectónica propuesta, permitiendo que la Empresa Eléctrica maneje sus procesos críticos de manera estructurada, flexible y alineada a las mejores prácticas de arquitectura moderna.

D) Arquitectura por capas

La arquitectura propuesta se estructura en un conjunto de capas claramente definidas, cuyo propósito es organizar las responsabilidades del sistema y asegurar un diseño coherente, mantenible y alineado con las necesidades operativas de la Empresa Eléctrica. Cada capa cumple un rol específico y se integra de manera controlada con las demás, permitiendo una evolución ordenada y una mejor gestión de los procesos de reclamos y reparaciones.

La Capa de Canal agrupa los puntos de interacción con los usuarios internos, como operadores del Call Center y personal técnico. Su función principal es ofrecer una experiencia de uso unificada, sin exponer la complejidad de los servicios subyacentes.

La Capa de Acceso incorpora componentes como el API Gateway y el Service Mesh, encargados de gestionar la autenticación, autorización, seguridad, enrutamiento y políticas de tráfico entre microservicios. Esta capa actúa como puerta de entrada controlada hacia todo el ecosistema de la arquitectura.

La Capa de Negocio coordina los procesos que requieren la participación de varios dominios. Aquí se implementan servicios de aplicación y orquestadores responsables de ejecutar casos de uso completos, gestionar Sagas, aplicar reglas transversales y asegurar la consistencia entre microservicios.

La Capa de Dominio contiene los microservicios definidos por bounded contexts, cada uno con su propia lógica de negocio, datos, reglas, eventos y API. Estos servicios son independientes entre sí y representan el núcleo funcional de la solución.

Finalmente, la Capa de Datos gestiona la persistencia a través de bases de datos o esquemas dedicados por microservicio, asegurando aislamiento, gobernanza y disponibilidad. Incluye también mecanismos como caching distribuido y almacenamiento auxiliar para reportes o auditorías.

E) Patrones arquitectónicos aplicados

La arquitectura propuesta se fundamenta en un conjunto de patrones arquitectónicos reconocidos que permiten construir una solución modular, escalable y alineada a los dominios de la Empresa Eléctrica. En primer lugar, se adopta el patrón de Microservicios, que divide la solución en servicios independientes basados en bounded contexts, facilitando el despliegue autónomo, la evolución modular y la separación clara de responsabilidades.

De manera complementaria, se aplica el patrón API Gateway, que actúa como punto único de entrada para los usuarios y sistemas, centralizando autenticación, autorización, enrutamiento y políticas de consumo. Este patrón evita la exposición directa de los microservicios y permite un control más eficiente del tráfico.

Asimismo, la propuesta incorpora el patrón de Arquitectura Orientada a Eventos (Event-Driven Architecture), permitiendo que ciertos procesos se comuniquen mediante publicación y consumo de eventos. Esto favorece el desacoplamiento entre servicios, mejora la escalabilidad y soporta escenarios de alta demanda o procesamiento asincrónico.

Finalmente, la solución se estructura sobre una Arquitectura por Capas, que organiza el sistema en niveles lógicos —canal, acceso, negocio, dominio y datos— garantizando orden, mantenibilidad y una separación clara entre responsabilidades técnicas y funcionales.

F) Atributos de calidad

El diseño arquitectónico propuesto se fundamenta en los siguientes atributos de calidad, esenciales para garantizar un funcionamiento sólido y eficiente en los procesos de reclamos y reparaciones de la Empresa Eléctrica:

- **Escalabilidad**: Permite que cada microservicio incremente su capacidad de manera independiente según la demanda operativa.
- **Mantenibilidad:** Facilita la evolución del sistema mediante componentes desacoplados, fáciles de modificar y extender.
- **Disponibilidad:** Asegura la continuidad del servicio incluso ante fallas parciales o picos de carga.
- **Seguridad:** Protege el acceso y maneja la información sensible mediante mecanismos centralizados y estándares modernos.
- **Observabilidad:** Proporciona métricas, logs y trazas que permiten monitorear el sistema y detectar problemas de forma oportuna.

6.3.2. Modelo C4 de la Arquitectura Propuesta

Diagrama de contexto (C1)

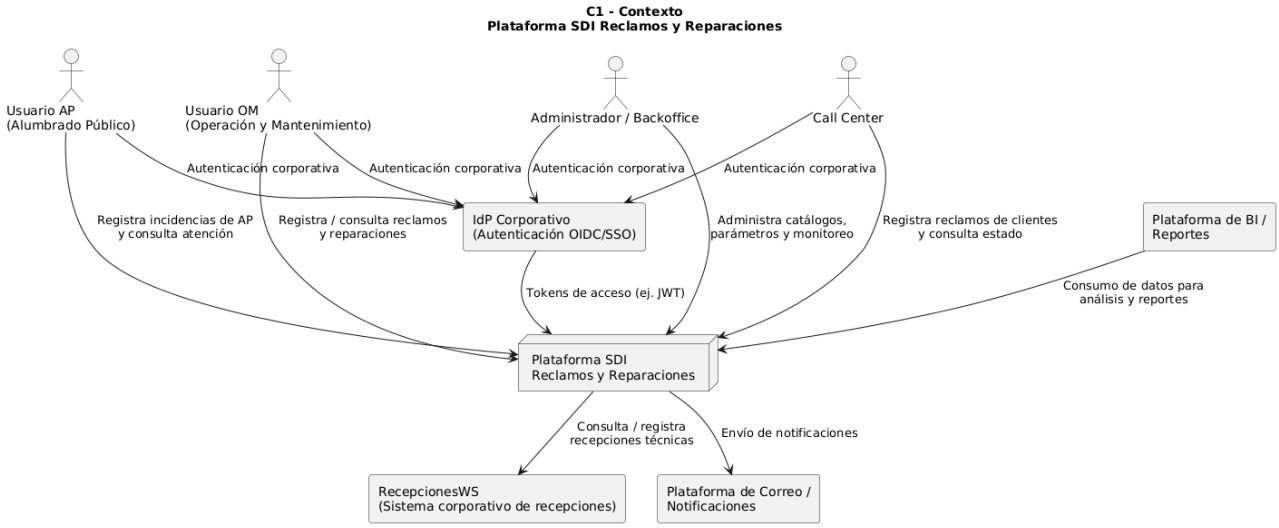

Figura 5. Diagrama de componentes Realizado por: Ing. Marco Clavijo - Ing. Amanda Cabascango

La arquitectura propuesta sitúa a la Plataforma SDI de Reclamos y Reparaciones como el sistema central que articula los principales procesos operativos de la Empresa Eléctrica. En este nivel de contexto, el sistema se visualiza como una unidad funcional única, enfocándose en las interacciones clave con los actores humanos y los sistemas corporativos externos.

Los actores internos: Usuario OM, Usuario AP, Call Center y Administrador acceden al sistema para registrar, consultar o gestionar reclamos y reparaciones. Estas interacciones se realizan a través de una aplicación web que canaliza las solicitudes hacia la plataforma, permitiendo que cada usuario ejecute las funciones alineadas a su rol operativo.

El sistema también mantiene integraciones críticas con plataformas externas. El IdP Corporativo provee autenticación centralizada mediante OIDC/SSO, garantizando un acceso seguro. El módulo se comunica con RecepcionesWS para consultar o registrar información técnica asociada a incidentes. La Plataforma de Correo y Notificaciones se utiliza para emitir avisos internos y externos, mientras que la Plataforma de BI consume datos consolidados con fines analíticos.

En conjunto, este modelo de contexto establece el perímetro funcional de la solución, sus actores principales y las interacciones esenciales con sistemas

externos, proporcionando una visión clara del ecosistema en el que opera la arquitectura propuesta.

Diagrama de contenedores (C2)

En el nivel de contenedores, la Plataforma SDI de Reclamos y Reparaciones se descompone en los componentes técnicos principales que dan soporte a la arquitectura propuesta basada en microservicios.

En el borde de la solución se ubica el Front-End Web SDI, una aplicación web (SPA) utilizada por los distintos tipos de usuario: Operación y Mantenimiento (OM), Alumbrado Público (AP), Call Center y Administrador. Este front-end canaliza todas las interacciones de los usuarios hacia la plataforma a través de llamadas HTTP/REST.

Como punto de entrada lógico, el API Gateway centraliza la autenticación, el enrutamiento de peticiones, la aplicación de políticas de seguridad y de tráfico. Este componente oculta la complejidad interna de los microservicios, expone APIs unificadas y se integra con el IdP Corporativo para la validación de tokens y el soporte de SSO/OIDC.

Por detrás del API Gateway se encuentra la Capa de Negocio u Orquestación, donde residen los servicios de aplicación que coordinan los flujos que involucran varios dominios: por ejemplo, un flujo completo de “registro de reclamo → generación de reparación → asignación de grupo trabajo → notificación”. Esta capa encapsula la lógica de orquestación, manejo de Sagas y reglas transversales sin mezclar la lógica propia de cada dominio.

El núcleo funcional se materializa en los Microservicios de Dominio:

- MS Reclamos: administra el ciclo de vida de los reclamos (registro, clasificación, seguimiento).
- MS Reparaciones: gestiona órdenes de trabajo, intervenciones y cierre de reparaciones.

- MS Catálogos Operativos: centraliza catálogos como motivos, zonas, materiales y otros elementos estructurales.
- MS Grupos de Trabajo: maneja disponibilidad, asignaciones y características del personal técnico.
- MS Notificaciones: centraliza el envío de notificaciones internas y hacia clientes.
- MS Integración Externa: encapsula la comunicación con sistemas corporativos, como RecepcionesWS y otras plataformas.

Cada microservicio persiste su información en una base de datos propia dentro de la Capa de Datos (BD de Reclamos, Reparaciones, Catálogos, Grupos de Trabajo, Notificaciones), alineado al principio de "base de datos por servicio". Esto favorece el desacoplamiento y la evolución independiente de cada dominio.

Finalmente, se representan las integraciones externas:

- El IdP Corporativo para autenticación centralizada.
- RecepcionesWS, consumido a través del MS de Integración Externa.
- La Plataforma de Correo/Notificaciones, utilizada por el MS de Notificaciones para el envío de mensajes.
- La Plataforma de BI, que consume datos desde los microservicios de Reclamos y Reparaciones para análisis y reportes.

En conjunto, este C2 refleja cómo la arquitectura propuesta pasa de una visión monolítica a una solución compuesta por contenedores bien definidos, alineados por dominio, con responsabilidades claras y preparados para escalar y evolucionar según las necesidades de la Empresa Eléctrica

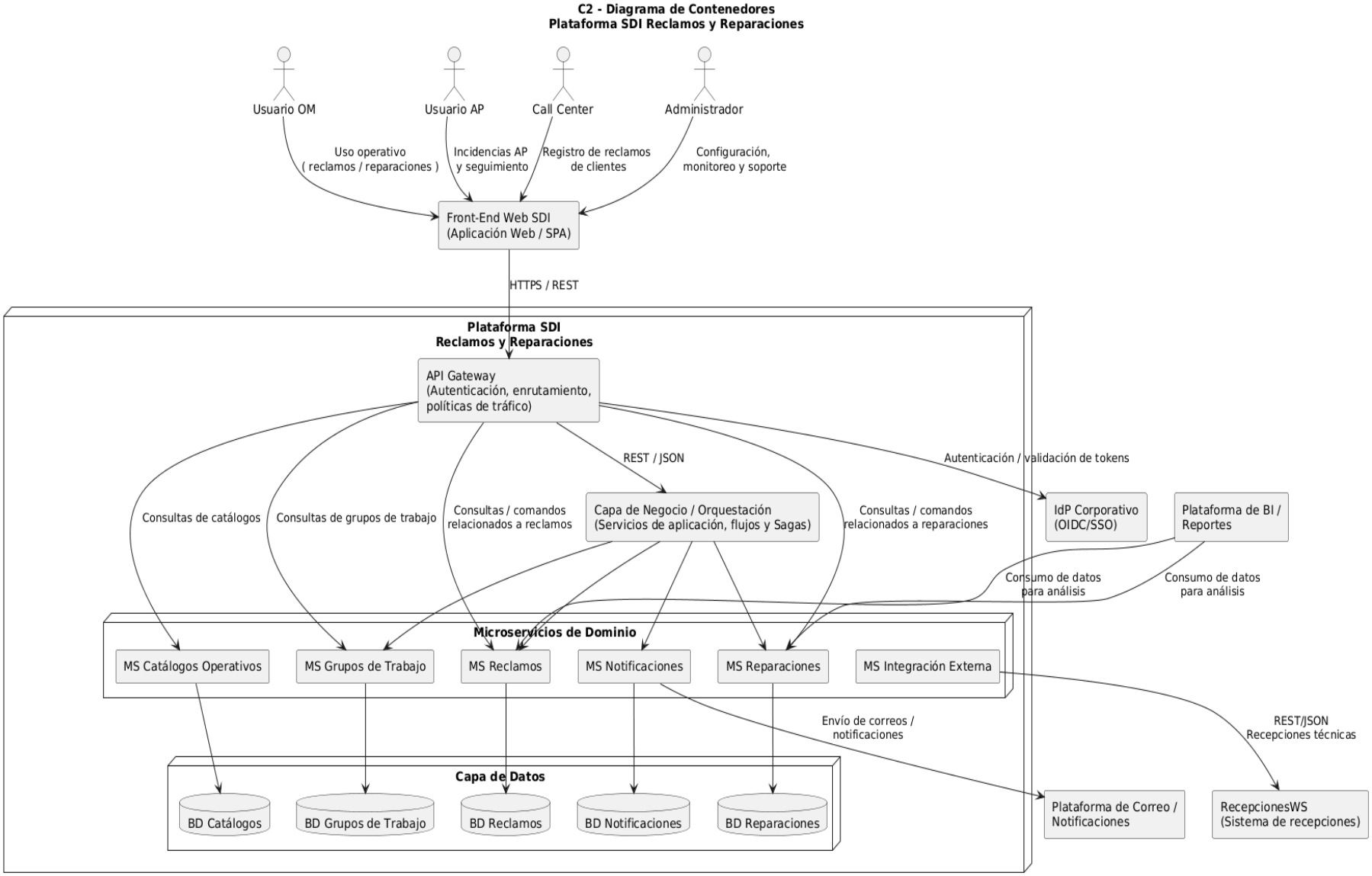
C2 - Diagrama de Contenedores
Plataforma SDI Reclamos y Reparaciones
Usuario OM
Usuario AP
Call Center
Administrador
Uso operativo
(reclamos / reparaciones)
Incidencias AP
y seguimiento
Registro de reclamos
de clientes
Configuración,
monitoreo y soporte
Front-End Web SDI
(Aplicación Web / SPA)
HTTPS / REST
Plataforma SDI
Reclamos y Reparaciones
API Gateway
(Autenticación, enrutamiento,
políticas de tráfico)
REST / JSON
Autenticación / validación de tokens
Consultas de catálogos
Consultas de grupos de trabajo
Consultas / comandos
relacionados a reclamos
Capa de Negocio / Orquestación
(Servicios de aplicación, flujos y Sagas)
Consultas / comandos
relacionados a reparaciones
IdP Corporativo
(OIDC/SSO)
Plataforma de BI /
Reportes
Consumo de datos
para análisis
Consumo de datos
para análisis
Microservicios de Dominio
MS Catálogos Operativos
MS Grupos de Trabajo
MS Reclamos
MS Notificaciones
MS Reparaciones
MS Integración Externa
Envío de correos /
notificaciones
REST/JSON
Recepciones técnicas
Capa de Datos
BD Catálogos
BD Grupos de Trabajo
BD Reclamos
BD Notificaciones
BD Reparaciones
Plataforma de Correo /
Notificaciones
RecepcionesWS
(Sistema de recepciones)

Figura 6. Diagrama de contenedores. Realizado por: Ing. Marco Clavijo - Ing. Amanda Cabascango

Diagrama de componentes (C3)

En el diagrama de componentes se detalla cómo está estructurada internamente la Plataforma SDI de Reclamos y Reparaciones, respetando la arquitectura por microservicios y separando claramente responsabilidades técnicas y de negocio.

En la parte superior se ubican dos componentes transversales:

- API Gateway: responsable de recibir las peticiones externas, aplicar autenticación y autorización, validar tokens, y enrutar las llamadas hacia las APIs REST de cada microservicio.
- Servicio de Orquestación de Casos de Uso: encapsula los flujos de negocio que atraviesan varios dominios, como el proceso completo desde el registro de un reclamo hasta la asignación de un grupo de trabajo y el envío de notificaciones.

Cada microservicio se descompone en componentes internos bien definidos:

Una API REST (por ejemplo, Reclamos REST API, Reparaciones REST API, GrupoTrabajo REST API), que expone los endpoints y traduce las solicitudes HTTP en comandos o consultas hacia la capa de aplicación.

Un Servicio de Aplicación (por ejemplo, ReclamosAppService, ReparacionesAppService, GrupoTrabajoAppService), que coordina las operaciones del caso de uso dentro del microservicio, invocando lógica de dominio, repositorios y, cuando aplica, publicación de eventos.

Un Dominio de Negocio por cada servicio (entidades, agregados y reglas específicas), donde reside la lógica de negocio propia de Reclamos, Reparaciones, Catálogos, Grupo de Trabajo y Notificaciones.

Un Repositorio por microservicio (por ejemplo, IReclamoRepository, IReparacionRepository, IGrupoTrabajoRepository), que encapsula el acceso a la base de datos y abstrae los detalles de persistencia.

Adicionalmente, los microservicios que participan en procesos asincrónicos cuentan con componentes dedicados para manejar eventos:

- Publicadores de eventos, que emiten notificaciones de cambios relevantes hacia el bus de mensajes.
- Suscriptores, que reaccionan a eventos provenientes de otros microservicios para ejecutar tareas dependientes.

La arquitectura se complementa con microservicios especializados como Integración Externa, orientado a encapsular las comunicaciones con servicios corporativos, y Notificaciones, que se encarga de la interacción con la plataforma de mensajería institucional.

En definitiva, este modelo arquitectónico representa una solución unificada y desacoplada que se alinea con los principios modernos de la arquitectura distribuida. La coordinación de microservicios, lograda mediante el API Gateway, una capa de orquestación y el bus de mensajes, establece una plataforma inherentemente estable, escalable y altamente coherente con los procesos críticos de reclamos y reparaciones. Esto asegura una base sólida y totalmente mantenible para la Empresa Eléctrica.

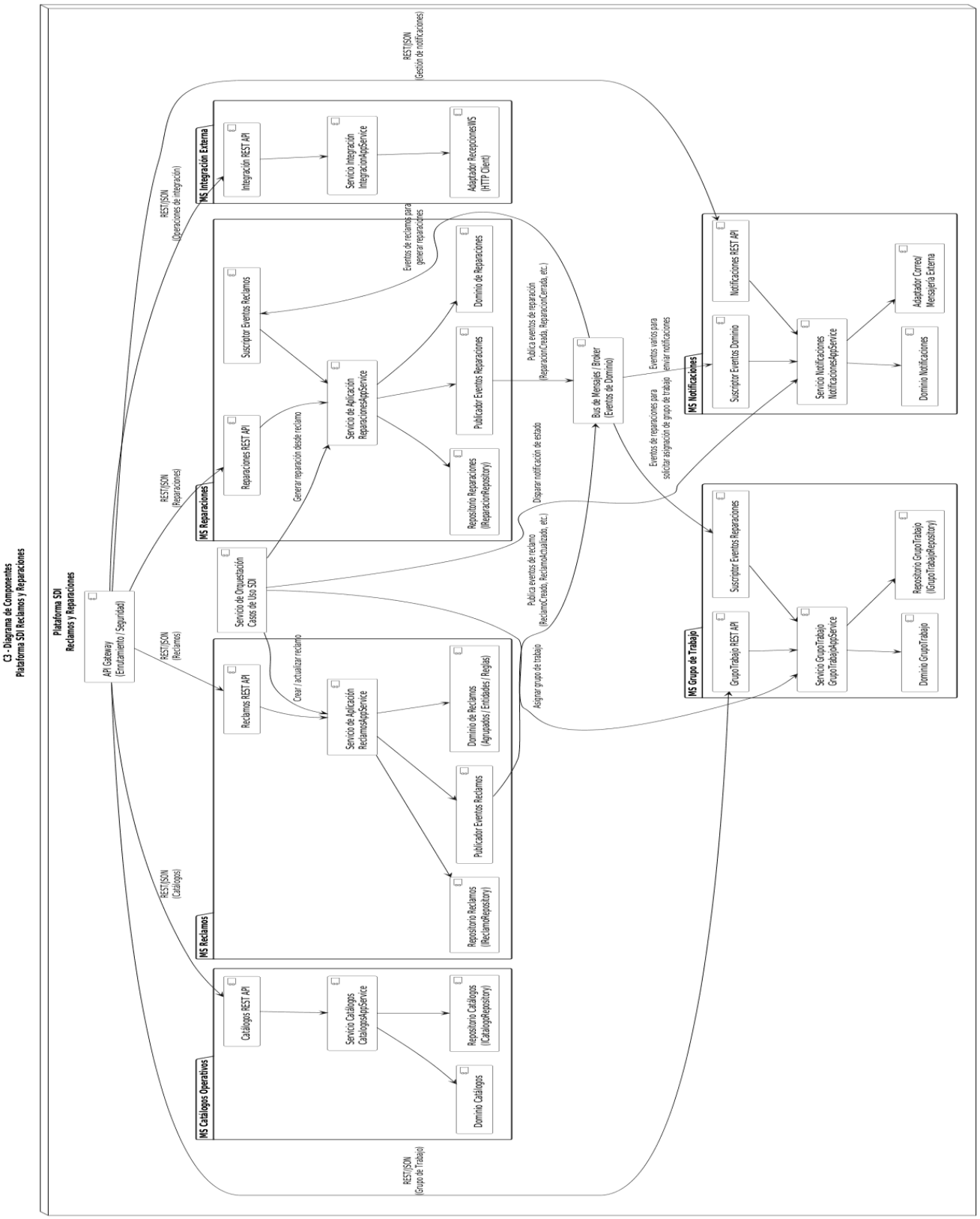

Figura 7. Diagrama de contenedores. Realizado por: Ing. Marco Clavijo - Ing. Amanda Cabascango

7. Resultados y discusión

7.1. Comparación entre Arquitectura Actual y Propuesta

La comparación entre la arquitectura actual y la propuesta revela diferencias abismales en estructura, capacidad operativa y potencial de crecimiento tecnológico. La estrategia vigente, anclada en un enfoque monolítico, concentra todas las funciones en un único artefacto. Esta consolidación genera un acoplamiento rígido que no solo dificulta la introducción de cualquier cambio, sino que ahoga la capacidad de escalar componentes de forma independiente. Dicha estructura propicia cuellos de botella, incrementa la complejidad del mantenimiento y afecta directamente la eficiencia en procesos críticos como reclamos y reparaciones, especialmente en escenarios de alta demanda.

En contraste, la arquitectura propuesta se sustenta en un modelo de microservicios que segmenta el sistema en componentes autónomos, cada uno ligado a un dominio específico del negocio. Este enfoque modular aporta la flexibilidad necesaria para que cada servicio evolucione sin comprometer la estabilidad global. La adopción de mecanismos clave como API Gateway, comunicación basada en eventos y REST, y el uso de bases de datos independientes para cada dominio, elimina la dependencia entre servicios y desbloquea la modernización continua del sistema.

Otro beneficio crucial es la mejora en la resiliencia, lograda porque los microservicios pueden ser escalados, desplegados y mantenidos de manera independiente, lo cual minimiza el riesgo de un colapso global. Esta arquitectura también integra, por diseño, mecanismos avanzados de observabilidad, seguridad y gestión de accesos. Estos elementos aseguran un control operativo más fino y garantizan la detección temprana de incidentes, ventajas de las que carece la solución vigente, lo que compromete su eficacia en contextos que demandan alta disponibilidad y respuesta ágil.

En conjunto, la comparación revela que la arquitectura basada en microservicios ofrece una alternativa más robusta, escalable y alineada a las necesidades futuras de la Empresa Eléctrica. Aunque aún no esta implementada, su análisis plantea una hoja de ruta clara para superar las restricciones del modelo monolítico y avanzar hacia una plataforma más flexible, mantenible y preparada para la evolución continua

7.2. Impacto de la Arquitectura Propuesta

La arquitectura de microservicios tiene la capacidad de transformar profundamente la operatividad del SDI. La plataforma no solo adquiere una mayor agilidad para absorber picos de demanda y facilitar la evolución tecnológica, sino que también respalda procesos críticos reduciendo al mínimo el riesgo de indisponibilidad. Este cambio genera un efecto directo en la organización: una operación mucho más estable, confiable y claramente orientada a la eficiencia.

7.2.1. Beneficios Operativos y Técnicos Esperados

La arquitectura propuesta resuelve problemas clave en la gestión de reclamos y reparaciones. Gracias a la distribución de la carga en microservicios, se anticipa una reducción considerable de los tiempos de respuesta al evitar los estrangulamientos inherentes al sistema monolítico. Esta estructura modular también garantiza una verdadera independencia de componentes, facilitando que se introduzcan mejoras y ajustes rápidamente sin desestabilizar el servicio global.

Asimismo, de acuerdo con el análisis técnico, la disponibilidad operativa se vería fortalecida al habilitar la capacidad de escalar microservicios específicos según demanda, reduciendo el riesgo de interrupciones totales durante picos de carga. Otro beneficio esperado radica en una utilización más eficiente de la infraestructura, dado que el escalamiento sería selectivo y orientado al consumo real de cada servicio, optimizando así los recursos.

Un beneficio clave es la incorporación de la observabilidad, que permite un monitoreo constante en tiempo real. Esto transforma la operación, ya que la

detección de fallas es temprana y la información obtenida es invaluable para las decisiones estratégicas. Por otra parte, la integración con otros sistemas corporativos será más robusta, gracias a la implementación de adaptadores dedicados y una capa de orquestación que estandariza los procesos, garantizando su coherencia y orden.

Finalmente, la seguridad se elevaría notablemente con la incorporación de la tokenización y la aplicación de controles de acceso específicos para cada servicio. Asimismo, el API Gateway actuaría como un punto unificado y centralizado de protección. Es importante aclarar que, aunque estos beneficios no han sido probados en un entorno real, están totalmente justificados por el análisis del diseño y la adherencia a las buenas prácticas de la arquitectura distribuida.

7.2.2. Limitaciones y Riesgos del Diseño

Si bien la arquitectura de microservicios ofrece importantes ventajas, es necesario reconocer que su implementación presenta desafíos considerables. Al tratarse únicamente de una propuesta conceptual, es crucial abordar estos elementos desde el principio. Los puntos que siguen reflejan las limitaciones y los riesgos inherentes identificados en nuestro análisis, además de delimitar con precisión qué aspectos escapan al alcance de esta investigación.

Limitaciones de la Propuesta

La propuesta arquitectónica, pese a su robustez conceptual, presenta ciertas limitaciones inherentes al enfoque:

- Mayor complejidad inicial: La transición hacia microservicios requiere una infraestructura más sofisticada, incluyendo mecanismos de despliegue, monitoreo distribuido y control de versiones entre servicios.
- Dependencia de capacidades tecnológicas avanzadas: La operación efectiva de la arquitectura demandaría herramientas complementarias como orquestadores, contenedores, gestores de configuración y mecanismos de observabilidad, que no forman parte de la plataforma actual.

- Aumento del tráfico interno: Los microservicios comunican frecuentemente entre sí, lo que incrementa la necesidad de gestionar latencias, balanceo de carga y calidad de servicio.
- Curva de aprendizaje para el personal técnico: Se requeriría capacitación para adoptar nuevos paradigmas como diseño por dominio, comunicación asincrónica, patrones de resiliencia y automatización de despliegues.
- Necesidad de gobernanza estricta: Para mantener consistencia entre dominios, contratos API y modelos de datos, sería necesario un marco de gobernanza que asegure la coherencia entre microservicios.

Estas limitaciones no invalidan la propuesta, pero sí representan consideraciones importantes para futuras etapas de evaluación e implementación.

Riesgos Durante la Migración

La migración desde un sistema monolítico hacia una arquitectura distribuida puede involucrar riesgos que deben gestionarse cuidadosamente:

- Interrupciones operativas si la transición no es gradual, especialmente al migrar componentes críticos como reclamos o reparaciones.
- Coexistencia temporal entre el monolito y los microservicios, lo que puede generar duplicidad de lógica o inconsistencias si no se aplican patrones como Strangler Fig.
- Incompatibilidad entre tecnologías actuales y las requeridas por microservicios, como la adopción de contenedores, mensajería o autenticación distribuida.
- Desalineación en procesos internos, ya que los usuarios deben adaptarse a cambios en funcionalidades o flujos operativos.
- Riesgos de dependencia con sistemas corporativos legados, que podrían no soportar nuevos volúmenes de comunicación o formatos de intercambio.

Estos riesgos reafirman la necesidad de que cualquier futura implementación sea planificada en fases y acompañada de mecanismos de validación continua.

Aspectos Fuera del Alcance del Proyecto

El análisis arquitectónico realizado se enfocó exclusivamente en los módulos de reclamos y reparaciones del SDI, por lo que ciertos aspectos se consideraron explícitamente fuera del alcance:

- Modernización o rediseño de sistemas corporativos externos, como RecepcionesWS, BI o plataformas de notificación.
- Renovación total de la infraestructura de TI, incluyendo migración a la nube o implementación de herramientas de orquestación.
- Reingeniería completa de procesos de negocio relacionados con operación y mantenimiento.
- Automatización CI/CD de la plataforma, que si bien podría ser necesaria en una futura implementación, no forma parte de esta etapa.
- Evolución o reemplazo del front-end existente, ya que la propuesta se centra únicamente en la capa de backend.
- Delimitar estos aspectos permite mantener claridad respecto al alcance analítico del proyecto y evita expectativas sobre elementos no abordados en esta fase.

La conclusión central de este capítulo es que la arquitectura de microservicios propuesta representa el camino más viable y sólido para transformar los módulos de reclamos y reparaciones de la Empresa Eléctrica. Si bien no se llevó a cabo una implementación práctica, el estudio prueba que este enfoque resuelve las limitaciones del sistema monolítico actual, entregando una escalabilidad mejorada, mayor resiliencia, una capacidad de integración superior y facilitando el mantenimiento futuro del software.

Asimismo, se identificaron beneficios operativos esperados, tales como tiempos de respuesta optimizados, mejor distribución de carga, observabilidad avanzada y mayor robustez en la seguridad y en la integración con sistemas corporativos. Estos resultados refuerzan la pertinencia de la propuesta y su alineación con las necesidades actuales y futuras de la organización.

Sin embargo, es necesario ser realistas: la transición a microservicios conlleva limitaciones importantes y riesgos intrínsecos. Esta realidad subraya la absoluta necesidad de abordar la adopción de la arquitectura mediante una planificación estratégica rigurosa, una sólida capacitación técnica y, sobre todo, un proceso de transición progresivo para mitigar cualquier potencial impacto operativo.

En resumen, este capítulo demuestra que la arquitectura basada en microservicios es una alternativa viable y con un enorme potencial de mejora para el SDI. Sin embargo, su puesta en marcha deberá llevarse a cabo en fases futuras, siempre bajo un enfoque rigurosamente controlado que contemple la visión integral de todos los sistemas y procesos institucionales.

8. Conclusiones

El desarrollo de esta investigación permitió analizar de manera integral las limitaciones presentes en la arquitectura actual del Sistema de Distribución de Información (SDI) de la Empresa Eléctrica, particularmente en los módulos de reclamos y reparaciones. A partir del diagnóstico realizado, se evidenció que el diseño monolítico, acoplado y carente de mecanismos modernos de escalabilidad, observabilidad y seguridad, dificulta la evolución del sistema y limita su capacidad de respuesta ante incrementos de demanda operativa.

La propuesta de arquitectura basada en microservicios se posiciona como la alternativa técnica viable y necesaria para superar estas barreras. El diseño excede la mera segmentación en dominios independientes, incorporando patrones arquitectónicos de vanguardia (como API Gateway, Event-Driven Architecture y arquitectura por capas). Esta estrategia aumenta sustancialmente la modularidad, resiliencia y mantenibilidad, materializándose en microservicios específicos dedicados a Reclamos, Reparaciones, Catálogos y Notificaciones, lo que define una estructura clara para el futuro del SDI.

Aunque la propuesta es conceptual y no se llevó a una implementación productiva, el análisis demuestra que este enfoque permitiría optimizar el uso de recursos y fortalecer la integración con sistemas corporativos. El enfoque microservicios es, en esencia, no solo viable, sino imprescindible. En definitiva, la transición hacia una arquitectura flexible y distribuida representa el paso fundamental para modernizar la plataforma tecnológica de la Empresa Eléctrica y asegurarle una respuesta sólida y consistente ante los desafíos del mañana.

A partir del análisis realizado, la primera y más importante recomendación es que cualquier avance hacia la implementación se aborde de manera progresiva y rigurosamente planificada. Un enfoque gradual es crucial, pues permite reemplazar el sistema monolítico de forma controlada, priorizando la continuidad operativa y minimizando riesgos. Esto implica integrar herramientas de vanguardia que son

esenciales en un entorno distribuido: plataformas de contenedorización, motores de orquestación, mecanismos de observabilidad avanzada y una gestión centralizada para accesos y seguridad.

La adopción exitosa de esta arquitectura requerirá que la organización fortalezca su gobernanza tecnológica y establezca lineamientos claros para la estandarización de APIs, la integración y la administración del ciclo de vida de los microservicios. Simultáneamente, la formación del personal técnico en buenas prácticas de arquitectura distribuida, diseño orientado a dominios y patrones de resiliencia es imprescindible para asegurar la evolución correcta y autónoma de la plataforma.

Mirando a la ejecución, se debe diseñar una estrategia integral que fusione los aspectos tecnológicos y organizacionales. La migración debe comenzar por los dominios más críticos del SDI para que los beneficios del nuevo modelo se perciban desde fases tempranas y se minimice la complejidad del proceso. Es clave complementar este esfuerzo con actividades de validación continua, pruebas de desempeño rigurosas y análisis de impacto detallados para asegurar la estabilidad del sistema durante toda la transición.

La organización también deberá fortalecer sus capacidades de supervisión y gobernanza, estableciendo mecanismos que permitan monitorear la evolución de la arquitectura, promover la consistencia entre servicios y asegurar el alineamiento con los objetivos institucionales. Además, será necesario prever planes de continuidad operativa, recuperación ante fallas y medidas de contingencia que soporten la operación tanto durante como después de la migración.

Estas recomendaciones buscan guiar a la institución hacia una implementación responsable, escalonada y técnicamente sustentada de la arquitectura propuesta. Con una planificación rigurosa, la transición hacia un modelo basado en microservicios permitirá consolidar una plataforma robusta, flexible y, lo más importante, preparada para afrontar los desafíos operativos y tecnológicos del futuro.

Referencias

Aghazadeh Ardebili, A. a. (s.f.). Digital Twins of smart energy systems: a systematic literature review on enablers, design, management and computational challenges. *Digital Twins of smart energy systems*, 94. Obtenido de https://doi.org/10.1186/s42162-024-00385-5

Auth0. (2023). *Active Directory / LDAP Connector*. Obtenido de https://auth0.com/docs/authenticate/identity-providers/enterprise-identity-providers/active-directory-ldap

Balalaie , A., Heydarnoori, A., & Jamshidi, P. (2016). *IEEE Software.* Obtenido de Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.

Barrett, C. (2020). *Observability Engineering.* O'Reilly Media.

Bass, L., Clements, P., & Kasman, R. (2012). *Software Architecture in Practice 3rd Edition.* Addison-Wesley.

Bengtsson, {. a. (24 de 11 de 2024). Architecture-level modifiability analysis ({ALMA}). *Journal of Systems and Software*, 129-147. Obtenido de https://linkinghub.elsevier.com/retrieve/pii/S0164121203000803

Brown, N. N. (2019). *Managing Technical Debt: Reducing Friction in Software Development.* Addison-Wesley.

Burns, B. &. (2022). *Cloud Native DevOps with Kubernetes (2nd ed.).*

Castillo Torres, V. a. (2021). Desarrollo y despliegue de una aplicacion web basada en microservicios.

CERT, S. (2022). *Secure Coding Standards. Carnegie Mellon University*. Obtenido de https://wiki.sei.cmu.edu

Cervantes Maceda, H., Velasco Elizondo, P., & Castro Careaga, L. (2016). *ARQUITECTURA DE SOFTWARE Conceptos y Ciclo de desarrollo.* Distrito Federal, México: Cengage Learning Editores.

Chen, L., & Muhammad , A. (2014). Towards an Evidence-Based Understanding of Emergence of Architecture through Microservices Adoption. Information and Software Technology.

Clemments, P. (2003). *Carnegie Mellon University.* Obtenido de Software architecture in practice: https://www.researchgate.net/profile/Rick-Kazman/publication/224001127_Software_Architecture_In_Practice/links/02bfe510fef5da3230000000/Software-Architecture-In-Practice.pdf

Corporation, O. (2022). *Oracle Forms Documentation Library*. Obtenido de https://docs.oracle.com/en/middleware/

Daniel and Maya, E. a. (20117). Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web.

Deshpande, A. (2016). *Modernizing Oracle Forms Applications.* Packt Publishing.

Di Francesco, P. a. (2018). *Migrating Towards Microservice Architectures: An Industrial Survey.* Obtenido de Migrating Towards Microservice Architectures: https://ieeexplore.ieee.org/abstract/document/8417114

Di Franceso, P., Lago, P., & Malavolta, I. (2018). *Architecting with microservices: A systematic mapping study.*

Dragoni, N., Giallorenzo, S., Lluch, A., Mazzarra, M., Montesi, F., Mustafin, R., & Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow.* Springer. doi:https://doi.org/10.1007/978-3-319-67425-4_12

Duan, H. a. (2023). *Construction of {Electricity} {Charge} {Information} {Management} {System} {Based} on {Network} {Microservice} {Technology}.* Ahmad, Ishfaq and Ye, Jun and Liu, Weidong. Obtenido de The 2021 {International.

Evans, E. (2003). *Domain Driven Design Tackling Complexity in the Heart of Software.* Obtenido de Academia: https://www.academia.edu/43965299/Domain_Driven_Design_Tackling_Complexity_in_the_Heart_of_Software_by_Eric_Evans_z_lib_org_

Fazio, M., Celesti, A., Villari, M., & Puliafito, A. (2020). *Microservices Architecture for Smart Energy Management. Future Generation Computer Systems.* Obtenido de https://doi.org/10.1016/j.future.2019.10.010

Ford, N. P. (2017). *Building Evolutionary Architectures: Support Constant Change. O'Reilly Media.*

Fowler, M. &. (2014). *Microservices*. Obtenido de martinfowler.com: https://martinfowler.com/articles/microservices.html

Fowler, M. (15 de 01 de 2014). *Bounded Context*. Obtenido de martinFowler.com: martinfowler.com/bliki/BoundedContext.html

Fowler, M. (2020). *Testing Strategies in Microservice Architectures*. Obtenido de https://martinfowler.com/articles/microservice-testing/

Fowler, M. (09 de 2025). *Blue Green Deployment*. Obtenido de martinFowler.com: https://martinfowler.com/bliki/BlueGreenDeployment.html

Galin, D. (2017). *Software Quality: Concepts and Practice (1st Edition).* doi:9781119134497

Hardt, D. (2012). *The OAuth 2.0 Authorization Framework. IETF.* Obtenido de https://datatracker.ietf.org/doc/html/rfc6749

Hightower, K. B. (2017). *Kubernetes: Up and Running: Dive into the Future of Infrastructure.* O'Reilly Media.

Humble, J. &. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley.

Hussain, F. H. (2021). Applications of blockchain and edge computing in smart grid: A review. IEEE Access, 9. Obtenido de https://doi.org/10.54097/jceim.v10i3.8704

IBM . (23 de 09 de 2021). *What are microservices?* Obtenido de https://www.ibm.com/cloud/learn/microservices

ISO. (2011). *ISO/IEC 27034-1: Information technology — Security techniques — Application security .*

Jimenez-Torres, V. H.-B.-P. (s.f.). Pattern Languages Software Architecture: An Approach To State of the Art. 4.

Kamlofsky, J. a. (2015). Un Enfoque para Disminuir los Efectos de los Ciber-ataques a las Infraestructuras Criticas. 2346-9927. doi:10.22517/23447214.8595

Kim, G. a. (2024, 12 01). *The-Devops-Handbook-Espanol.pdf.* Retrieved from https://www.docdroid.net/F6xjVuX/the-devops-handbook-espanol-pdf

Kim, G. D. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution.

Kindervag, J. (2010). *Build Security Into Your Network's DNA: The Zero Trust Network Architecture.* Forrester Research.

Kruchten, P. N. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 18-21. doi:10.1109/MS.2012.167

Kruchten, P., Ozkaya, I., & Nord, R. (2012). *Managing Technical Debt: Reducing Friction in Software Development.* Pearson Education.

Lewis, J. a. (01 de 12 de 2024). *Microservices*. Obtenido de martinfowler.com: https://martinfowler.com/articles/microservices.html

López, D. a. (s.f.). Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web. *Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web*. Obtenido de https://dspace.redclara.net/bitstream/10786/1277/1/93%20Arquitectura%20de%20Software%20basada%20en%20Microservicios%20para%20Desarrollo%20de%20Aplicaciones%20Web.pdf

Luis, d. V. (2016). *Valencia: Conferencia de Directores y Decanos de Ingenieria informatica .*

Marco de Trabajo para Seleccionar un Patrón Arquitectónico en el Desarrollo de Software. (2021). *Revista Ibérica de Sistemas e Tecnologias de Informação*, 568-581. Obtenido de https://www.proquest.com/docview/2562269745/abstract/CAF76075084F4AFFPQ/7

Martini, A. B. (2018). Managing technical debt in software-intensive products. . *Journal of Systems and Software*, 69-82. doi:10.1016/j.jss.2016.05.018

Merkel, D. (2014). *Docker: lightweight Linux containers for consistent development and deployment.* Linux Journal.

Microsoft Azure Architecture Center . (2023). *Microservice architecture style*. Obtenido de Architecture Center: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Monolitos vs. Microservicios en Arquitectura de Software: Perspectivas para un Desarrollo Eficiente. (24 de 11 de 2024). *Memorias de las JAIIO, 10*(5), 42-54. Obtenido de https://revistas.unlp.edu.ar/JAIIO/article/view/17984

Morandini, M. a. (2021). Considerations about the efficiency and sufficiency of the utilization of the Scrum methodology: A survey for analyzing results for development teams. *Computer Science Review*. doi:https://doi.org/10.1016/j.cosrev.2020.100314

NERC. (2023). *Critical Infrastructure Protection Standards. North American Electric Reliability Corporation*. Obtenido de https://www.nerc.com

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* 1005 Gravenstein Highway North, Sebastopol, CA: O'Reilly Media Inc. Recuperado el 02 de 2025, de https://github.com/rootusercop/Free-DevOps-Books-1/blob/master/book/Building%20Microservices%20-%20Designing%20Fine-Grained%20Systems.pdf

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media.* doi:978-1492047841

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems.* "O'Reilly Media, Inc.

Ortega, M. (2020). Tecnologias para arquitecturas basadas en microservicios: Patrones y soluciones para aplicaciones desplegadas en contenedores. Obtenido de

https://www.amazon.com/-/es/Tecnolog%C3%ADas-para-arquitecturas-basadas-microservicios/dp/620041131X

OWASP. (2023). *OWASP Secure Coding Practices. Open Web Application Security Project*. Obtenido de https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/

Pérez, G. a. (06 de 09 de 2018). *Reingeniería de aplicaciones Oracle Forms legadas a partir de reglas de extracción de elementos, y migración a tecnología Java por generación automática de código.* Obtenido de https://riunet.upv.es/handle/10251/106731

Qumer Gill, A. a. (2018). DevOps for information management systems. *VINE Journal of Information and Knowledge Management Systems*, 122-139.

Richards, M. &. (2020). *Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media.*

Richards, M. (2015). *Software Architecture Patterns.* 1005 Gravenstein Highway North, Sebastopol, CA: O' Reilly Media. Recuperado el 03 de 2025, de Software Architecture Patterns: https://github.com/chapin666/books/blob/master/architecture/software-architecture-patterns.pdf

Richardson, C. (2019). *Microservices Patterns: With examples in Java.* Manning. ISBN: 9781617294549

Rodríguez, C. a. (12 de 05 de 2021). *Model-based assisted migration of oracle forms applications: The overall process in an industrial setting.* doi:https://doi.org/10.1002/spe.2981

Rozanski, N., & Woods, E. (2012). *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives.* Massachussets: Pearson Education Inc.

Sato, D. (09 de 2025). *Canary Release*. Obtenido de https://martinfowler.com/bliki/CanaryRelease.html

Schwaber, K. &. (2020). *The Scrum Guide*. Obtenido de https://scrumguides.org/

Siriwardena, P., & Dias, N. (2019). *Microservices Security in Action.* Manning Publications.

Sommerville, I. (2020). *Software Engineering (10ª ed.).* Pearson.

Sonatype. (2023). *State of the Software Supply Chain.* Obtenido de https://www.sonatype.com/state-of-the-software-supply-chain

Swift, D. (25 de 09 de 2024). *Java Microservices in the Energy Sector: A Case Study*. Obtenido de Java Microservices in the Energy Sector: https://www.springfuse.com/energy-sector-microservices-architecture/

Trebejo, L. a. (17 de 12 de 2024). *Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios*. Obtenido de Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios: https://cybertesis.unmsm.edu.pe/item/aa46756c-3576-4627-b157-0389da41df09

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Casallas, R. (10 de 2015). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. (págs. 583-590). Bogota, Colombia:

Institute of Electrical and Electronics Engineers (IEEE). doi:10.1109/ColumbianCC.2015.7333476

Wolpers, S. (2020). *Scrum Anti-Patterns: A Handbook for Agile Practitioners.* (Primera Edición ed.). LeanPub.

Zampetti, F. a. (2021). CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 471-482. doi:https://doi.org/10.1109/ICSME52107.2021.00048

Zapata, C. M. (2009). Revisión de la literatura en interoperabilidad entre sistemas heterogeneos de software. *29*(2), 42-47.ISSN: 0129-5608