UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERÍA MECÁNICA ELÉCTRICA, ELECTRÓNICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

EVALUACIÓN DE LA COHESIÓN ENTRE COMPONENTES BACKEND BASADOS EN MICROSERVICIOS MEDIANTE LA APLICACIÓN DE ESTRATEGIAS DE DESACOPLAMIENTO EN LA ENTIDAD FINANCIERA LOS ANDES

TESIS

PRESENTADA POR:

ALEX FREDY ESCALANTE MARON

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO DE SISTEMAS

PUNO – PERÚ

2024

Reporte de similitud

NOMBRE DEL TRABAJO

EVALUACIÓN DE LA COHESIÓN ENTRE C OMPONENTES BACKEND BASADOS EN MICROSERVICIOS MEDIANTE LA APLICA CIÓN DE ESTRATEGIAS DE DESACOPLA MIENTO EN LA ENTIDAD FINANCIERA L OS ANDES

AUTOR

ALEX FREDY ESCALANTE MARON

RECUENTO DE PALABRAS

28390 Words

RECUENTO DE CARACTERES

165826 Characters

RECUENTO DE PÁGINAS

152 Pages

TAMAÑO DEL ARCHIVO

3.5MB

FECHA DE ENTREGA

Jan 26, 2024 2:14 PM GMT-5

FECHA DEL INFORME

Jan 26, 2024 2:16 PM GMT-5

18% de similitud general

El total combinado de todas las coincidencias, incluidas las fuentes superpuestas, para cada base c

- 17% Base de datos de Internet
- 2% Base de datos de publicaciones
- Base de datos de Crossref
- Base de datos de contenido publicado de Crossr
- 7% Base de datos de trabajos entregados

Excluir del Reporte de Similitud

- Material bibliográfico
- Material citado
- Material citado
- Coincidencia baja (menos de 8 palabras)

UNA PUNO

Firmado digitalmente por FERNANDEZ CHAMBI Mayenka FAU 20145496170 soft
Motivo: Soy el autor del documento
Fecha: 26.01.2024 14:22:32 -05:00

V°B°

Firmado digitalmente por SOTOMAYOR ALZAMORA Guina Guadalupe FAU 20145496170 hard
Motivo: Doy V°B°
Fecha: 26.01.2024 15:52:11 -05:00

Resumen

DEDICATORIA

Con una gratitud sincera y una profunda emoción que late como el pulso de la sabiduría, dedico la culminación de mi tesis a aquellos que han sido faros inquebrantables en mi travesía académica. A mis amados padres, Enrique Escalante y Lourdes Maron, les tributo mi más sincero reconocimiento por su dedicación incansable y esfuerzo perseverante, fuerzas telúricas que han impulsado cada paso en este arduo sendero del conocimiento. Extiendo asimismo mi aprecio a mis queridas hermanas, Mirian y Mariluz, por sus alientos constantes. Con un sentido profundo de cariño y agradecimiento, comparto este logro como una celebración de la fortaleza y unidad.

Alex Fredy Escalante Maron

AGRADECIMIENTOS

En primer lugar, deseo expresar mi profundo agradecimiento a Dios, cuya guía y fortaleza han sido la luz que ha iluminado cada paso de esta travesía académica.

A mi familia, les dedico un especial reconocimiento por su apoyo incondicional a lo largo de este arduo camino.

Asimismo, extiendo mi reconocimiento a los distinguidos docentes de la Escuela Profesional de Ingeniería de Sistemas por impartir una formación profesional sólida y enriquecedora.

A mis amigos y superiores en Los Andes, mi gratitud por su apoyo generoso durante la realización de esta investigación.

Expreso mi sincero reconocimiento y profunda gratitud hacia mi asesora, la M.Sc. Mayenka Fernández Chambi, por su orientación admirable y experta, así como por su generosa contribución de tiempo y paciencia a lo largo de todo el proceso académico e investigativo.

Finalmente, agradezco a los distinguidos jurados por sus valiosas observaciones, sugerencias y correcciones que han enriquecido este trabajo. Su contribución ha sido de gran relevancia para la calidad y rigor de esta investigación.

Alex Fredy Escalante Maron

ÍNDICE GENERAL

Pág.

DEDICATORIA

AGRADECIMIENTOS

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ANEXOS

ÍNDICE DE ACRÓNIMOS

RESUMEN 15

ABSTRACT 16

CAPÍTULO I

INTRODUCCIÓN

1.1. PLANTEAMIENTO DEL PROBLEMA 18

1.2. FORMULACIÓN DEL PROBLEMA 19

1.2.1. Problema general 19

1.2.2. Problemas específicos 19

1.3. JUSTIFICACIÓN DE LA INVESTIGACIÓN 20

1.4. OBJETIVOS DE LA INVESTIGACIÓN 21

1.4.1. Objetivo general 21

1.4.2. Objetivos específicos 22

1.5. HIPÓTESIS 22

1.5.1. Hipótesis general 22

CAPÍTULO II

REVISIÓN DE LITERATURA

2.1. ANTECEDENTES DE LA INVESTIGACIÓN 23

2.1.1. Antecedentes internacionales 23

2.1.2. Antecedentes nacionales 28

2.2. MARCO TEÓRICO 30

2.2.1. Arquitectura de software 30

2.2.2. Componentes de software 33

2.2.2.1. Componentes de procesamiento de datos 35

2.2.2.2. Componentes de almacenamiento de información 35

2.2.2.3. Componentes basados en la interacción del usuario 35

2.2.3. Evolución arquitectónica 36

2.2.4. Descomposición arquitectónica 38

2.2.5. Arquitecturas monolíticas 38

2.2.6. Arquitectura de microservicios 43

2.2.7. Calidad de software 55

2.2.7.1. La familia de normas ISO/IEC 25000 56

2.2.7.2. ISO/IEC 25010 56

2.2.8. Principios SOLID 57

2.2.8.1. Principio de responsabilidad única 59

2.2.8.2. Principio abierto/cerrado 60

2.2.8.3. Principio de sustitución de liskov 61

2.2.8.4. Principio de segregación de interfaces 62

2.2.8.5. Principio de inversión de dependencias 63

2.2.9. Clean architecture 64

2.2.10. Cohesión 67

2.2.11. Acoplamiento 69

CAPÍTULO III

MATERIALES Y MÉTODOS

3.1. UBICACIÓN GEOGRÁFICA DE ESTUDIO 71

3.2. OPERACIONALIZACIÓN DE VARIABLES 72

3.3. DISEÑO Y MÉTODO DE LA INVESTIGACIÓN 73

3.3.1. Tipo de diseño 73

3.3.2. Diseño de investigación 73

3.3.3. Método 74

3.4. POBLACIÓN Y MUESTRA 76

3.4.1. Población 76

3.4.2. Muestra 76

3.5. MATERIALES Y EQUIPOS UTILIZADOS 77

3.6. MÉTODO PARA LA RECOPILACIÓN DE LOS DATOS 79

3.7. MÉTODOS PARA EL ANÁLISIS DE LOS DATOS 79

3.7.1. Identificación de microservicios utilizando descomposición funcional . 79

3.7.2. Medición de la cohesión y el acoplamiento 82

3.7.2.1. Métrica de cohesión de clases 83

CAPÍTULO IV

RESULTADOS Y DISCUSIÓN

4.1. DESCRIPCIÓN DEL CASO DE ESTUDIO 89

4.2. OBJETIVO ESPECÍFICO 01, IDENTIFICACIÓN DE MICROSERVICIOS USANDO DESCOMPOSICIÓN FUNCIONAL 90

4.3. OBJETIVO ESPECÍFICO 02, DISEÑO DE LA ARQUITECTURA DE MICROSERVICIOS 99

4.3.1. Capa de dominio 100

4.3.2. Capa de aplicación 102

4.3.3. Capa de infraestructura 103

4.3.4. Capa de presentación 105

4.3.5. Integración del modelo arquitectónico en los microservicios 106

4.4. OBJETIVO ESPECÍFICO 03, EVALUACIÓN DE LA COHESIÓN DE LOS COMPONENTES BACKEND EN MICROSERVICIOS 108

4.4.1. Desarrollo de una aplicación web para la evaluación de cohesión 108

4.4.1.1. Tecnologías utilizadas 109

4.4.1.2. Funcionalidades implementadas 112

4.4.1.3. Resultados de la implementación de la aplicación web 113

4.4.1.4. Evaluación de la métrica de cohesión de clases 117

4.4.2. Evaluación de la cohesión en los microservicios de Credirapp 120

4.4.2.1. Proceso de evaluación de microservicios 121

4.4.2.2. Resultados tras la evaluación de cohesión 125

V. CONCLUSIONES 135

VI. RECOMENDACIONES 138

VII. REFERENCIAS BIBLIOGRÁFICAS 140

ANEXOS 144

ÁREA : Ingeniería de software, bases de datos e inteligencia de negocios

TEMA: Arquitectura de Microservicios

FECHA DE SUSTENTACIÓN: 31 de enero del 2024

ÍNDICE DE FIGURAS

Pág.

Figura 1 Vista de componentes y conectores en un sistema de software. 34

Figura 2 ISO/IEC 25000. 56

Figura 3 ISO/IEC 25010 Características de calidad. 57

Figura 4 Principios de diseño S.O.L.I.D. 58

Figura 5 Una clase con una única responsabilidad. 60

Figura 6 Diagrama Integrado de Onion Architecture. 65

Figura 7 Ubicación geográfica de estudio. 71

Figura 8 Procedimiento metodológico. 74

Figura 9 Pasos del método de descomposición funcional. 80

Figura 10 Ejemplo de Grafo de dependencia entre Operaciones / Relaciones. 82

Figura 11 Clase compuesta por métodos y atributos. 84

Figura 12 Árbol de subconjuntos representativo de los grupos de la clase. 85

Figura 13 Diagrama de casos de uso identificados para el sistema Credirapp. 90

Figura 14 Gráfico de dependencia de operación/relación del sistema. 96

Figura 15 Grupos Identificados como Microservicios. 97

Figura 16 Modelo de Arquitectura para el desarrollo del sistema. 100

Figura 17 Representación de Onion Architecture en cada Microservicio. 107

Figura 18 Procedimiento para la implementación de la aplicación web. 109

Figura 19 Interfaz para la gestión de proyectos. 114

Figura 20 Interfaz para la creación de proyectos y nodos. 115

Figura 21 Opciones del nodo. 115

Figura 22 Interfaz para la relación por dependencia entre nodos. 116

Figura 23 Representación visual de dependencias. 117

Figura 24 Ejemplo de clase compuesta por métodos y atributos..... 118

Figura 25 Resultados del ejemplo de cohesión planteado por Saadati & Motameni. . 119

Figura 26 Representación de los siete Microservicios generados en la aplicación. 122

Figura 27 Arquitectura de Capas del Microservicio “Clientes”. 122

Figura 28 Clases de las Capas del Microservicio “Clientes”. 123

Figura 29 Representación de métodos y atributos del Microservicio “Clientes”. 124

Figura 30 Porcentaje de cohesión en el Microservicio de Clientes. 126

Figura 31 Porcentaje de cohesión en el Microservicio de Créditos..... 127

Figura 32 Porcentaje de cohesión en el Microservicio de Simulador. 128

Figura 33 Porcentaje de cohesión en el Microservicio de Riesgo Crediticio. 130

Figura 34 Porcentaje de cohesión en el Microservicio de Externos. 131

Figura 35 Porcentaje de cohesión en el Microservicio de Archivos. 132

Figura 36 Porcentaje de cohesión en el Microservicio de Catálogos. 133

ÍNDICE DE TABLAS

Pág.

Tabla 1 Principios S.O.L.I.D. 59

Tabla 2 Matriz de consistencia del proyecto. 72

Tabla 3 Módulos del sistema Credirapp para el proceso de admisión de créditos. 77

Tabla 4 Tabla de procesos funcionales. 92

Tabla 5 Tabla de grupos de datos. 93

Tabla 6 Matriz de Operación / Relación del sistema. 94

Tabla 7 Microservicios identificados para el sistema. 98

Tabla 8 Aspectos Técnicos de la implementación de la aplicación web. 110

Tabla 9 Resumen de resultados de la cohesión de clase. 120

ÍNDICE DE ANEXOS

Pág.

ANEXO 1: Procedimiento para la aplicación de la métrica de cohesión 144

ANEXO 2: Diagrama de navegación de la aplicación Credirapp 149

ANEXO 3: Capturas de pantalla de la aplicación móvil Credirapp..... 150

ANEXO 4: Declaración jurada de autenticidad de tesis 151

ANEXO 5: Autorización para el depósito de tesis en el Repositorio Institucional 152

ÍNDICE DE ACRÓNIMOS

API: Application Programming Interface / Interfaz de Programación de Aplicaciones.

COSMIC: Common Software Measurement International Consortium / Consortio Internacional Común de Medición de Software.

DevSecOps: Development, Security, and Operations / Desarrollo, Seguridad y Operaciones.

DISC: Dominio, Infraestructura, Seguridad, Calidad.

HTTP: Hypertext Transfer Protocol / Protocolo de Transferencia de Hipertexto.

IEC: International Electrotechnical Commission / Comisión Electrotécnica Internacional.

IDE: Integrated Development Environment / Entorno de Desarrollo Integrado.

ISO: International Organization for Standardization / Organización Internacional de Normalización.

LSP: Liskov Substitution Principle / Principio de Sustitución de Liskov.

OCP: Open/Closed Principle / Principio Abierto/Cerrado.

OWASP: Open Web Application Security Project / Proyecto de Seguridad en Aplicaciones Web Abiertas.

RCC: Reporte crediticio de clientes.

REST: Representational State Transfer / Transferencia de Estado Representacional.

RPC: Remote Procedure Call / Llamada a Procedimiento Remoto.

SAMM:	Software Assurance Maturity Model / Modelo de Madurez de Seguridad en Aplicaciones.
SBS:	Superintendencia de banca y seguro.
SPOF:	Single Points of Failure / Puntos Únicos de Fallo.
SOA:	Service-Oriented Architecture / Arquitectura Orientada a Servicios.
SQuaRE:	Software product Quality Requirements and Evaluation / Evaluación y Calidad del Software.
SRP:	Single-Responsibility Principle / Principio de Responsabilidad Única.
SOLID:	Conjunto de Principios SRP, OCP, LSP, ISP y DIP.
TDD:	Test-Driven Development / Desarrollo Guiado por Pruebas.

RESUMEN

En el dinámico ecosistema financiero, caracterizado por una demanda incesante de servicios digitales, la evolución de las infraestructuras tecnológicas se convirtió en una imperativa necesidad para las instituciones bancarias. La transición a arquitecturas ágiles busca proactivamente mejorar la adaptabilidad y escalabilidad. En este contexto, la presente investigación surge con el propósito de abordar los desafíos que enfrenta la entidad financiera Los Andes, debido a la naturaleza de sus sistemas monolíticos. La problemática central residía en el notable grado de acoplamiento presente en estos sistemas, lo que generó la necesidad imperante de evaluar y perfeccionar la cohesión entre los componentes Backend, específicamente en la aplicación Credirapp. La propuesta de solución se enfocó en la adopción estratégica de Microservicios, buscando potenciar la eficiencia y modularidad del sistema. En este contexto, se planteó tres objetivos fundamentales: identificar Microservicios significativos mediante descomposición funcional, modelar una arquitectura de Microservicios que optimice la cohesión y minimice el acoplamiento, y evaluar la cohesión de los componentes Backend. La metodología aplicada combina la Descomposición Funcional para la identificación, la implementación de la arquitectura Onion Architecture para el modelo, y la métrica de Saadati y Motameni para la evaluación. Se logró una evaluación de la cohesión general del 88.5%, calificándose como Muy Alta, y se identificaron con precisión siete Microservicios claves cuya arquitectura maximiza la cohesión y minimiza el acoplamiento, además se desarrolló una aplicación web para evaluar la cohesión basada en la métrica de Saadati y Motameni que puede ser utilizada en futuras investigaciones.

Palabras Clave: Cohesión, Desacoplamiento de componentes, Mejoras de software, Microservicios.

ABSTRACT

In the dynamic financial ecosystem, characterized by an incessant demand for digital services, the evolution of technological infrastructures has become an imperative necessity for banking institutions. The transition to agile architectures proactively seeks to enhance adaptability and scalability. In this context, the present research emerges with the purpose of addressing the challenges faced by the financial entity "Los Andes," due to the nature of its monolithic systems. The central issue resided in the notable degree of coupling present in these systems, which generated the urgent need to assess and improve cohesion between Backend components, specifically in the Credirapp application. The proposed solution focused on the strategic adoption of Microservices, aiming to enhance system efficiency and modularity. In this context, three fundamental objectives were proposed: identify significant Microservices through functional decomposition, model a Microservices architecture that optimizes cohesion and minimizes coupling, and evaluate the cohesion of Backend components. The applied methodology combines Functional Decomposition for identification, the implementation of Onion Architecture for the model, and the Saadati and Motameni metric for evaluation. A general cohesion evaluation of 88.5% was achieved, qualifying as Very High. Seven key Microservices were accurately identified whose architecture maximizes cohesion and minimizes coupling. Additionally, a web application was developed to evaluate cohesion based on the Saadati and Motameni metric, which can be used in future research.

Key Words: Cohesion, Component decoupling, Microservices, Software improvements.

CAPÍTULO I

INTRODUCCIÓN

En el vertiginoso escenario financiero actual, donde cada fracción de tiempo es esencial, las organizaciones se enfrentan al desafío monumental de gestionar una abrumadora cantidad de datos y procesos en tiempo real. Este desafío se acentúa con los constantes cambios y avances en el mercado. A pesar de su robustez, las estructuras monolíticas convencionales a menudo experimentan limitaciones notables en escalabilidad y adaptabilidad.

Los sistemas de software en la financiera Los Andes adoptan una estructura monolítica, integrando diversos componentes y funciones interconectadas. No obstante, esta elección presenta desafíos significativos debido a una baja cohesión, es decir los componentes del sistema están altamente acoplados, complicando el mantenimiento, escalabilidad y adaptación del sistema frente a las cambiantes demandas y avances tecnológicos.

En este contexto, emerge una solución de gran relevancia: la arquitectura de Microservicios, unidades modulares que no solo facilitan una mejora continua ágil, sino que también proporcionan una eficiente implementación de actualizaciones, los Microservicios a su vez guardan una fuerte relación con la cohesión. La cohesión se refiere a la relación lógica y funcional entre los diferentes Microservicios que componen el sistema. Una alta cohesión implica que los Microservicios están diseñados de manera coherente y se centran en cumplir una única responsabilidad.

La presente investigación tuvo el propósito principal de evaluar la cohesión en arquitecturas de Backend basadas en Microservicios de la aplicación Credirapp, sistema

cuya función principal consiste en gestionar el proceso de admisión de créditos. Para lograrlo, se planteó tres objetivos específicos: identificar elementos fundamentales de desacoplamiento mediante la aplicación de método específico de descomposición funcional de Tyszberowicz et al (2018), modelar una arquitectura de Microservicios con el objetivo de reforzar la independencia y escalabilidad de cada Microservicio, y por último evaluar la cohesión de los componentes resultantes mediante métrica propuesta por Saadati & Motameni (2014). Este enfoque permitió no solo optimizar la interconexión y la flexibilidad del sistema, sino también asegurar un rendimiento óptimo y una mantenibilidad mejorada en entornos distribuidos.

1.1. PLANTEAMIENTO DEL PROBLEMA

La entidad financiera Los Andes utiliza sistemas de software monolíticos, los cuales incorporan una variedad de componentes y funciones interconectadas. Sin embargo, estos sistemas presentan desafíos sustanciales debido a un acoplamiento considerable, lo que complica la tarea de mantenimiento, escalabilidad y adaptación del sistema a las cambiantes demandas y avances tecnológicos.

En este contexto, surgió la necesidad de comprender la importancia de los Microservicios y la cohesión en dicha entidad. La cohesión refiere a la relación lógica y funcional entre los diferentes Microservicios que componen el sistema. Una alta cohesión implica que los Microservicios están diseñados de manera coherente y se centran en cumplir una única responsabilidad, lo cual facilita el proceso de mantenimiento, evolución y escalabilidad de manera significativa.

El problema radica en que una entidad financiera con baja cohesión y una arquitectura monolítica puede presentar desafíos significativos. Una estructura monolítica puede dificultar la implementación de nuevas funcionalidades, ya que cualquier cambio

en una parte del sistema puede tener efectos no deseados en otras áreas. Además, la falta de modularidad y la dependencia entre componentes pueden obstaculizar la evolución del sistema, ralentizando los procesos de desarrollo y limitando la capacidad de adaptación a nuevas tecnologías y requerimientos del mercado.

Además, la adopción de una arquitectura basada en Microservicios con una alta cohesión puede generar beneficios significativos. Los Microservicios independientes permiten un desarrollo y despliegue más ágil, lo que facilita la implementación de nuevas funcionalidades y la corrección de errores de manera más rápida. Además, al tener una responsabilidad única, los Microservicios se vuelven más fáciles de mantener y escalar de manera individual, sin afectar al sistema en su totalidad.

Por lo tanto, la relevancia de los Microservicios y la cohesión en un sistema financiero es fundamental e imprescindible, ya que una arquitectura adecuada puede impulsar la eficiencia, la flexibilidad y la adaptabilidad de la organización en un entorno financiero en constante evolución.

1.2. FORMULACIÓN DEL PROBLEMA

1.2.1. Problema general

¿Cuál estrategia de desacoplamiento aplicar en componentes Backend basados en Microservicios para evaluar la cohesión entre sus elementos, con el propósito de mejorar la eficiencia y escalabilidad del sistema?

1.2.2. Problemas específicos

- ¿Cuáles son los elementos de desacoplamiento más relevantes en el patrón arquitectónico de Microservicios?

- ¿Cómo diseñar y modelar una arquitectura de Backend basada en Microservicios, con el fin de lograr una arquitectura modular, escalable y eficiente?
- ¿Cómo evaluar la cohesión de los componentes resultantes del Backend, con el propósito de asegurar una arquitectura de alta calidad y eficiencia?

1.3. JUSTIFICACIÓN DE LA INVESTIGACIÓN

En la actualidad, la entidad financiera Los Andes utiliza sistemas de software monolíticos que incluyen diferentes componentes y servicios interconectados entre sí. Sin embargo, estos sistemas presentan problemas de fuerte acoplamiento, lo que dificulta su mantenimiento, escalabilidad y evolución; por ejemplo, una mayor complejidad y dificultad para entender y depurar el código del sistema, dificultad para realizar pruebas, diagnóstico de errores y dificultad para introducir cambios o mejoras en el sistema. También aumenta el tiempo y los costos asociados a la implementación de nuevas características o funcionalidades.

Para solucionar estos problemas, se propuso descomponer los componentes para el Backend en arquitecturas de Microservicios para la aplicación Credirapp, este sistema tiene como función principal la gestión del proceso de admisión de créditos. Con esta estrategia, se busca mejorar la cohesión del sistema, y reducir la interdependencia entre sus diferentes componentes.

La cohesión se refiere al grado en que los elementos de un módulo de software están relacionados entre sí y trabajan juntos para lograr un objetivo común. En otras palabras, se trata de la medida en que las partes de un módulo están unidas y son interdependientes para cumplir con su propósito. Una alta cohesión indica que los elementos del módulo están altamente relacionados y trabajan juntos de manera efectiva,

mientras que una baja cohesión indica que los elementos del módulo no están bien relacionados y pueden no estar cumpliendo su propósito de manera eficiente. Por ello, la investigación tuvo como objetivo medir la relación entre los métodos públicos en función de su contribución relativa a la funcionalidad pública global de una clase (Saadati & Motameni, 2014). Para determinar la contribución relativa de los métodos públicos, se ha utilizado un concepto de árbol de subconjuntos. Dicha métrica mide la cohesión de la clase en un intervalo de 0 a 1.

La relevancia de esta investigación se evidencia en la posibilidad de que la entidad financiera Los Andes optimice la cohesión de sus componentes, impulsando mejoras significativas en eficiencia, escalabilidad y adaptabilidad a las dinámicas cambiantes del mercado. Este logro se alcanzará mediante la implementación de una arquitectura de Microservicios más modular y desacoplada. Además, el estudio ha propiciado una organización y estructuración mejoradas del código, facilitando así su mantenimiento y evolución a largo plazo.

En síntesis, esta investigación ha abordado una problemática relevante para la entidad financiera Los Andes, ofreciendo una solución innovadora y eficaz para potenciar la eficiencia y escalabilidad de sus sistemas de software. La aplicación de estrategias de desacoplamiento en arquitecturas de Microservicios no solo se presenta como una solución beneficiosa para la entidad en cuestión, sino también como un aporte valioso para otras empresas u organizaciones que enfrenten desafíos similares.

1.4. OBJETIVOS DE LA INVESTIGACIÓN

1.4.1. Objetivo general

Aplicar estrategias de desacoplamiento en componentes Backend basados en Microservicios, con el fin de evaluar la cohesión entre sus componentes.

1.4.2. Objetivos específicos

- Identificar los elementos de desacoplamiento más significativos dentro del patrón arquitectónico de Microservicios utilizando descomposición funcional.
- Modelar una arquitectura de Backend basada en Microservicios, con el fin de lograr una arquitectura modular, escalable y eficiente.
- Evaluar la cohesión de los componentes resultantes del Backend mediante la aplicación de los principios de cohesión en Microservicios.

1.5. HIPÓTESIS

1.5.1. Hipótesis general

La aplicación de las estrategias de desacoplamiento en Microservicios Backend permite evaluar adecuadamente la cohesión entre sus componentes, en la entidad financiera Los Andes.

CAPÍTULO II

REVISIÓN DE LITERATURA

2.1. ANTECEDENTES DE LA INVESTIGACIÓN

2.1.1. Antecedentes internacionales

Orjuela Velandia (2022), en su proyecto "*Descomposición de componentes Frontend de tipo web mediante estrategias de desacoplamiento en arquitecturas de Microservicios*", este se basó en hallar métodos de descomposición de software, esencialmente en los componentes de Frontend, que ayudaran a solucionar problemas y mejorar las características esenciales de las aplicaciones. Por lo tanto, el propósito del proyecto de investigación consistió en emplear una técnica de desacoplamiento fundamentada en el modelo arquitectónico de Microservicios, con el fin de evaluar el grado de cohesión alcanzado por los componentes resultantes. A través de la métrica de Saadati & Motameni (2014). Con el objetivo de cumplir este propósito, se proporcionó una visión general sobre la descomposición arquitectónica y los componentes de software, así como una descripción detallada de los mecanismos empleados para la descomposición de estos. Adicionalmente, se exhibe la aplicación práctica utilizada como modelo de referencia, la cual está enfocada en el sector financiero. Por último, el modelo resultante de la descomposición es altamente exitoso logrando una evaluación de cohesión del 55.8%, ya que no solo cumple con las funcionalidades establecidas en la etapa de diseño, sino que también muestra características cohesivas según la evaluación efectuada. Además, cada interacción emergente del proceso de desacoplamiento garantiza una respuesta oportuna a las

demandas del usuario y permite explorar configuraciones que de otra forma no serían posibles.

Pedraza Coello (2021), en su investigación *"Método DISC: Separando sistemas en Microservicios"*, se presenta una solución para ayudar a los arquitectos de software a determinar qué funcionalidad y datos deben ser modelados juntos en un mismo Microservicio. Se ha desarrollado un método específico para abordar este problema, el cual ha sido analizado en detalle. La propuesta se basa en la hipótesis inicial y tiene como objetivo principal la separación de sistemas en Microservicios para mejorar su desacoplamiento. El método propuesto utiliza los requisitos funcionales, considerando su nivel de detalle en el proceso funcional, para tomar decisiones sobre su agrupación dentro de un mismo microservicio y para determinar qué grupos de datos son persistentes en cada uno de ellos. Por ende, el método propuesto aquí se fundamenta en el concepto de proceso funcional introducido por COSMIC, de manera que la granularidad de un microservicio se establece según el número de procesos funcionales que lo componen. Este enfoque se apoya en la descomposición funcional. En el desarrollo de esta investigación, se consiguió identificar cinco microservicios mediante la aplicación del método DISC. Este método se basa en un conjunto de criterios de acoplamiento, está organizado en cinco etapas que aborda perspectivas de dominio, infraestructura, seguridad, calidad y la resolución de contradicciones. En la literatura, se han referenciado estos criterios de acoplamiento como factores importantes que influyen en la toma de decisiones para diseñar arquitecturas basadas en Microservicios. Además, según el método propuesto, se sugiere la utilización de conceptos del estándar internacional de

medición de tamaño funcional de software COSMIC para medir tanto la granularidad de los Microservicios como el nivel de acoplamiento entre estos.

López Hinojosa (2017), en su proyecto de investigación titulado *"Arquitectura de software basado en Microservicios para desarrollo de aplicaciones web de la asamblea nacional"*, el objetivo fue descubrir las tecnologías, metodologías y arquitecturas utilizadas por la Coordinación General de Tecnologías de la Información y Comunicación para desarrollar aplicaciones web, además de identificar las tecnologías disponibles para la creación y despliegue de Microservicios. Para llevar a cabo la investigación se utilizó una metodología cualitativa con enfoque descriptivo y diseño documental. Se aplicó la técnica de grupo focal a los funcionarios responsables del desarrollo de software en la Asamblea Nacional y se llevó a cabo una revisión bibliográfica sobre arquitectura de Microservicios. Se validó la arquitectura propuesta utilizando el método de análisis de concesiones mutuas de arquitectura. El análisis realizado permitió identificar la situación actual de los Microservicios y su implementación, así como los requisitos y necesidades para el desarrollo de aplicaciones web y cómo abordarlos mediante el diseño de una arquitectura de software. El estudio también presentó un marco de trabajo y la introducción de nuevas prácticas de desarrollo de aplicaciones centradas en servicios con características como resiliencia y escalabilidad, entre otras, que se derivan de la implementación de Microservicios.

Nebel (2018), *"Arquitectura de Microservicios para plataformas de integración"*, esta investigación propone la incorporación de sistemas heterogéneos a través del uso de Plataformas de Integración, que son infraestructuras especializadas diseñadas para facilitar la comunicación entre

sistemas al proporcionar mecanismos para detectar y solucionar incompatibilidades entre ellos. Con el surgimiento de la computación en la nube, se han presentado nuevos desafíos y necesidades para las Plataformas de Integración en términos de escalabilidad y eficiencia. En este contexto, la arquitectura de Microservicios ofrece ventajas como la capacidad de escalamiento independiente y la facilidad de mantenimiento, lo que podría mejorar positivamente el rendimiento de las Plataformas de Integración en diversos escenarios. En esta investigación se analiza la viabilidad de aplicar la arquitectura de Microservicios en la construcción de las Plataformas de Integración. Se estudia el impacto que tendría esta arquitectura y se determinan los escenarios en los cuales sería adecuado su uso. Asimismo, se presentan opciones de arquitectura y diseño para la creación de Plataformas de Integración basadas en Microservicios, las cuales son evaluadas considerando varios factores. Se propone en el estudio una metodología para evaluar el impacto de la arquitectura de Microservicios en los atributos de calidad. Se obtienen dos resultados principales del análisis, uno que describe cómo se ven afectadas las Plataformas de Integración en función de sus atributos de calidad y otro que determina cómo afecta la implementación de la arquitectura de Microservicios a la calidad de la plataforma. Siguiendo con el tema, se identifican una serie de rasgos de los contextos de integración, que facilitan la evaluación de sí un escenario se beneficia al implementar una API basada en Microservicios. Con base en las sugerencias de diseño y arquitectura, se presenta una propuesta principal que es adaptable a distintos escenarios, y también variantes que son aplicables a situaciones específicas. La valoración de la propuesta principal se fundamenta en tres factores: i) la aplicación de patrones y buenas prácticas de Microservicios, ii) los atributos de calidad de la Plataforma

de Integración, y iii) la creación de un prototipo. Como resultado de esta investigación, se concluye que es posible utilizar la arquitectura de Microservicios para la construcción de Plataformas de Integración en escenarios que cumplen ciertas características, y que las propuestas de arquitectura basadas en esta metodología son factibles.

Moreno Bernal (2022), en su proyecto de investigación "Modelo de evolución arquitectónica de monolito a Microservicios para el sistema de información ISys de la dirección nacional de admisiones de la universidad nacional de Colombia", en este trabajo de investigación de maestría, se llevó a cabo un estudio de caso que describe la transformación y evolución de un sistema de información con arquitectura monolítica a una basada en Microservicios. El estudio abarcó la creación, implementación y validación de un modelo de evolución que se basa en un análisis detallado de la arquitectura del sistema y en los requisitos operativos y técnicos de la Dirección Nacional de Admisiones de la Universidad Nacional de Colombia. El modelo de evolución desarrollado en esta tesis proporciona una forma de establecer prioridades en la evolución del sistema mediante el análisis de dependencias ascendentes y el impacto previsto de dicha evolución. También se proponen adaptaciones a tres patrones de arquitectura que se utilizan como estrategia principal para la evolución iterativa e incremental del sistema, sin interrumpir su operación total o parcial, o las dependencias asociadas. En conclusión, el modelo propuesto facilita la evolución arquitectónica del sistema de información ISys, pasando de un enfoque monolítico a uno basado en Microservicios de manera progresiva. Esto se logra manteniendo la disponibilidad del sistema y conservando los datos históricos sin interrumpir la operación habitual de la organización ni comprometer la seguridad del sistema. El modelo

de evolución resultante es altamente flexible, ya que su implementación se lleva a cabo de manera iterativa, lo que permite priorizar servicios. Al inicio de cada iteración, se actualiza la matriz de dependencias del sistema para lograr una mejor priorización de los servicios.

Nieto Sánchez (2017), "*Modelo de Arquitectura de Software para Aplicaciones iOS basado en Clean Architecture*", se orienta hacia el diseño y aplicación de un modelo de arquitectura de software específicamente destinado a aplicaciones iOS. Basándose en la filosofía de Clean Architecture de Robert C. Martin, el estudio realiza una revisión exhaustiva del estado actual del campo y analiza modelos existentes. El resultado es un modelo evolutivo que se adapta de manera precisa a los objetivos planteados, destacando la importancia de la modularidad, reutilización, la implementación de patrones de diseño y los principios SOLID. Respaldado por una metodología derivada del Test Driven Development (TDD), o Desarrollo Guiado por Pruebas en español. El modelo logra simplificar el proceso de desarrollo, garantizar la calidad del software, facilitar su mantenimiento y expansión, y optimizar el tiempo empleado en desarrollo y refactorización. Este trabajo no solo contribuye significativamente al ámbito del desarrollo de software para iOS, sino que también ofrece un marco sólido que puede ser aplicado en investigaciones futuras y en situaciones prácticas dentro de este contexto específico.

2.1.2. Antecedentes nacionales

Burgos Díaz (2021), en su estudio *"Destrucción de un monolito en Microservicios en fandango Latinoamérica"*, esta investigación tiene como objetivo principal explorar el desacoplamiento de las funcionalidades de un

software, dividiéndolo en varios programas independientes, con el propósito de optimizar el rendimiento y la eficiencia para la empresa Fandango Latinoamérica. La razón detrás de esta decisión fue el aumento en la cantidad de usuarios en el software, lo que resultó en una sobrecarga en el servidor y elevaba los costos de capacidad computacional necesarios para evitar que el sistema colapsara. Esto a su vez llevó a costos elevados que la empresa no podía sostener. El propósito de la iniciativa es implementar progresivamente Microservicios en el sistema de Fandango para mejorar la eficiencia en la gestión y administración de los recursos tecnológicos y financieros asignados a cada servicio. La estrategia empleada consiste en identificar las interdependencias entre los servicios y comenzar la migración desde los servicios que no tienen dependencias hasta los servicios que tienen una alta interdependencia con otros. Se empleó la metodología Scrum para manejar el proyecto, la estrategia de programación Test Driven Development, y la estrategia de escalamiento de la infraestructura de auto-escalamiento predictivo. Los resultados obtenidos se traducen en una plataforma altamente disponible, una reducción del 36,05% en costos y una arquitectura distribuida en Microservicios.

Machuca Pajuelo (2021), en su investigación *"Implementación de la práctica DeVSecOps aplicada a Microservicios para una entidad financiera"*, en este trabajo se detalla la implementación de la práctica DevSecOps para Microservicios en una entidad bancaria, con el fin de validar la madurez de seguridad en el ciclo de desarrollo de software y mejorar las capacidades tecnológicas para asegurar la seguridad del software durante todo el proceso de desarrollo. La implementación de DevSecOps se centró en la utilización del Framework SAMM v2.0 de OWASP, el cual permitió la integración de la seguridad en un entorno DevOps ya consolidado y alineado con los objetivos

estratégicos de la organización. Gracias a la aplicación de metodologías de gestión de proyectos, se logró crear una propuesta comercial sólida y ejecutar el proyecto con la calidad deseada, la metodología SAMM de OWASP desempeñó un papel fundamental en el desarrollo del proyecto, ya que permitió alinear los objetivos de la entidad bancaria con los lineamientos y las mejores prácticas en cuanto a la seguridad de las aplicaciones.

2.2. MARCO TEÓRICO

Como elemento fundamental en el desarrollo de este proyecto, resulta crucial comprender a detalle la idea que se pretende investigar mediante la definición precisa de los términos utilizados. En consecuencia, varios de los conceptos utilizados en este estudio son establecidos a partir de fuentes bibliográficas y referencias académicas confiables.

2.2.1. Arquitectura de software

La arquitectura de software es un subcampo específico de la ingeniería de software que se dedica al diseño estructural de los sistemas de software, lo que facilita una comprensión completa de los aspectos funcionales y no funcionales del sistema. La arquitectura de software consiste en una representación conceptual que convierte los objetivos de las organizaciones en sistemas de software, mejorando la comunicación entre los arquitectos y las partes interesadas. Esto aporta claridad en cuanto a los atributos necesarios, tanto funcionales como de calidad y cómo se logran los objetivos (Paul Clements et al., 2010).

La arquitectura de software no se limita a ser una etapa concreta en el proceso de desarrollo, sino que está presente a lo largo de todas las fases de vida del software. Se refleja en cada etapa del desarrollo y comprende las decisiones

de diseño fundamentales que afectan la calidad del producto de software. Hay que considerar que la arquitectura desde las etapas iniciales suele tener un efecto positivo en la calidad del software (Paul Clements et al., 2010).

Es válido considerar que la arquitectura de software inicia su influencia en la fase de diseño, pero no es preciso equiparar la totalidad de la etapa de diseño con la arquitectura de software. Las decisiones arquitectónicas son cruciales para el sistema, y cualquier modificación en la arquitectura tiene un impacto significativo en el costo. La falta de un diseño arquitectónico en un sistema no implica que el sistema carezca de una arquitectura. Uno de los principios fundamentales de la arquitectura de software sostiene que todos los sistemas tienen una arquitectura, lo que significa que todos los sistemas pueden ser examinados y estudiados desde una perspectiva arquitectónica (Richard N. Taylor et al., 2009).

La arquitectura de software engloba un conjunto definido de decisiones críticas para el diseño, las cuales se deben aplicar en el desarrollo y construcción de sistemas de software con una alta calidad. Cuando se habla de un sistema de software de alta calidad, se refiriere a que cada una de estas decisiones influyen en las relaciones entre sus componentes, es necesario definir reglas claras de cómo los elementos arquitectónicos deben estar organizados con el propósito de cumplir una labor específica, y proporcione una comprensión de cómo deben ser los comportamientos de cada uno de los procesos incluidos, establezca límites en las decisiones relacionadas con la interacción de los elementos y en última instancia, defina los detalles de implementación de toda la solución de manera que cumplan plenamente con las especificaciones requeridas (Richard N. Taylor et al., 2009).

Del mismo modo, estas decisiones deben orientarse a la mejora estructural y la composición de los elementos del sistema, a través de una interpretación precisa de los requisitos funcionales y no funcionales, Cuando hacemos mención de una estructura de software, se consideran cada una de las partes que conforman el sistema y la forma de integración de todo el conjunto de elementos de software (J. McC Smith & D. Stotts, 2002). Por lo tanto, es importante destacar que la arquitectura de un sistema también se puede definir en términos de sus componentes, con esto refiriéndose a la estructura del software, la cual comprende elementos, propiedades visibles, relaciones que permiten su comunicación y sus principios de diseño y evolución. Dada esta definición se encapsulan dos conceptos esenciales, el término de estructura y el término de elementos, son pilares fundamentales para comprender lo que implica la creación de una arquitectura (J. McC Smith & D. Stotts, 2002).

Asimismo, se observa que la arquitectura está ligada fuertemente con el software. Ya que, para definir un conjunto específico de atributos y características, es esencial considerar el panorama completo de sus posibles interacciones y como estas contribuyen a las interacciones para el correcto funcionamiento del sistema (Gamma, 2002). Esta idea se alinea con el concepto de que la arquitectura de software permite realizar y obtener la abstracción de un sistema a través de sus componentes, sus propiedades y las relaciones entre estos. Simultáneamente, considerando que la arquitectura de software está vinculada con su adecuada integración de elementos (Anubha Sharma et al., 2015).

2.2.2. Componentes de software

Los componentes de software se definen como el conjunto de todos los elementos que tienen presencia respecto al tiempo de ejecución computacional, dichos elementos se refieren a los procesos, los objetos, los clientes y servidores, las bases de datos, entre otros. Por lo tanto, cuando nos referimos a un componente de software se precisa objetivamente en aquellas unidades que conforman un sistema bajo ciertos lineamientos arquitectónicos, que por lo general establecen un objetivo específico. Estas piezas de software poseen un nivel de abstracción mucho más elevado que cualquier elemento funcional dentro del conjunto, ya que se perciben como módulos que exponen un comportamiento relacional, más allá del desempeño de sus labores asignadas (Paul Clements et al., 2010).

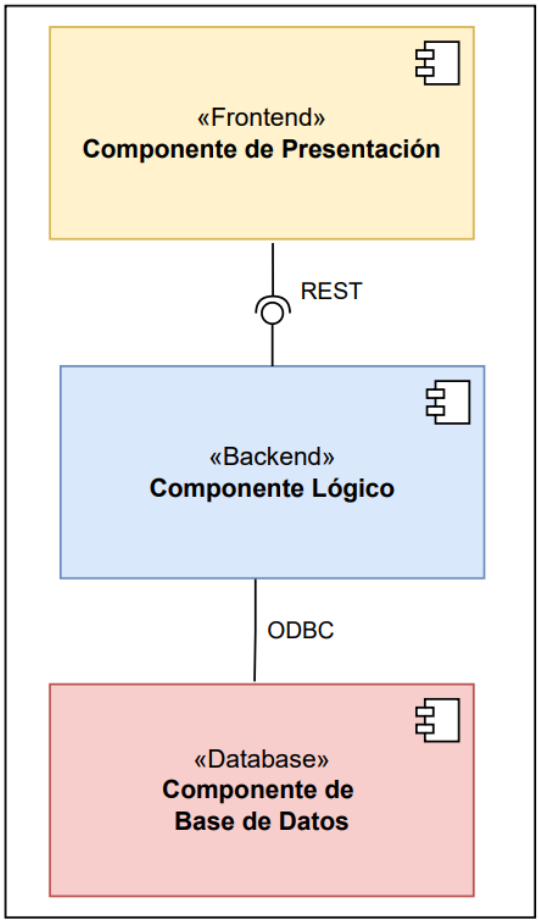
En la mayoría de los casos, los componentes de software ofrecen una serie de servicios que son utilizados por otros elementos del sistema, esto posibilita que el conjunto trabaje de forma integral, por lo tanto, cuando uno de estos componentes es removido, cierta funcionalidad o fragmento de software deja de operar (Addison-Wesley, 2012).

Asimismo, cuando se desarrolla un componente de software en consonancia con una arquitectura definida, se sigue un proceso de dos pasos. En primer lugar, se busca definir y especificar una colección de responsabilidades que este elemento debe cumplir. En segundo lugar, se establecen una serie de tareas que deben cumplirse, independientemente de la naturaleza del componente. De manera análoga, la arquitectura de software determina y define cómo debe comportarse y relacionarse el componente dentro del sistema, ya que parte de las configuraciones preliminares vienen de la mano con el diseño definido en las

etapas introductorias de la construcción. En consecuencia, estos lineamientos son los que establecen los límites en términos de funcionalidad, rendimiento, detalles de implementación y su relación con las otras partes del sistema (S. Mahmood 1 et al., 2007).

Figura 1

Vista de componentes y conectores en un sistema de software.



Fuente: (Orjuela Velandia, 2022)

2.2.2.1. Componentes de procesamiento de datos

Estos tipos de componentes, ocasionalmente denominados componentes de Backend, tienen la responsabilidad de controlar y gestionar toda la lógica de negocios dentro de un sistema de software. Allí, se llevan a cabo el procesamiento de la información, con el propósito de ofrecer una serie de servicios que proporcionan los datos necesarios para cumplir con funcionalidades específicas. Estos componentes están clasificados como parte de los elementos que se encargan del procesamiento y tratamiento de datos, ya que precisamente su función principal en el contexto del software consiste en recibir datos de entrada, realizar una transformación interna y exponer otro tipo de información que es útil y beneficioso para el resto de los elementos del sistema (Christopher Newman et al., 2018).

2.2.2.2. Componentes de almacenamiento de información

Los componentes de almacenamiento se centran de manera más específica en comparación con los otros componentes de software previamente descritos, ya que su principal responsabilidad es conservar la información transmitida desde otros componentes. En este contexto, no se realiza ningún tipo de procesamiento adicional, salvo por la posible conversión de texto, fechas o datos numéricos (Christopher Newman et al., 2018).

2.2.2.3. Componentes basados en la interacción del usuario

Dentro de este amplio conjunto de elementos se encuentran los elementos de Frontend, que son componentes de software visibles para el usuario. Estos engloban todas las interfaces visuales o mecánicas que interactúan directamente con el usuario final a través de una pantalla, un receptor o un actuador.

Concretamente, los componentes de Frontend abarcan todas las interfaces, plataformas, medios interactivos o elementos gráficos que posibilitan la visualización y muestran un conjunto de datos o información al usuario (Godbolt, 2016).

Las propiedades de los componentes de Frontend incluyen aspectos tanto inherentes a la construcción del sistema como características exclusivas de este tipo de componentes, como el tipo de interacción que el usuario tiene con la interfaz. Además, existen propiedades directamente relacionadas con el diseño y desarrollo del componente, como un rendimiento apropiado, la eficiencia en la operación de la interfaz, y consideraciones no funcionales contempladas durante la fase de análisis del sistema (Godbolt, 2016).

2.2.3. Evolución arquitectónica

La evolución arquitectónica de los sistemas de software se relaciona con las modificaciones que experimentan los componentes, las conexiones y las características. Por lo general, el diseño y la estructura de los sistemas de software se modifican con el tiempo para adaptarse a correcciones y nuevos requisitos. Las razones detrás de la evolución de una arquitectura pueden variar, y algunas de las más habituales incluyen factores como la interpretación errónea o incorrecta de los requisitos, así como cambios rápidos en el modelo de negocio (Richard N. Taylor et al., 2009), cumplir con nuevas o modificar propiedades no funcionales del sistema y la necesidad de adaptarse a nuevos contextos operativos (Rozanski & Woods, n.d.).

En la literatura, este concepto puede ser denominado como evolución, transformación, mantenimiento, adaptación, y otros términos, pero en su esencia

se refiere al proceso común de rediseñar y reestructurar arquitecturas. Para llevar a cabo la evolución de una arquitectura de manera exitosa, es esencial contar con un profundo entendimiento del sistema; de lo contrario, la calidad del diseño se verá comprometida y tenderá a deteriorarse con el tiempo (Richard N. Taylor et al., 2009).

No obstante, las empresas se encuentran con el desafío de lidiar con una alta rotación de empleados, lo cual también afecta a los equipos de tecnología. Los sistemas que experimentan una pérdida de calidad a lo largo de su evolución, pero que siguen siendo esenciales y desempeñan un papel fundamental en la organización, a menudo se denominan sistemas heredados o Legacy Systems (Richard N. Taylor et al., 2009).

En la actualidad, numerosas organizaciones buscan evolucionar muchos sistemas heredados con un propósito particular, ya sea migrándolos a otras arquitecturas o actualizándolos dentro de la misma arquitectura. Aunque la idea de reconstruir un sistema desde cero pueda parecer atractiva en el proceso, hace hincapié en que no es la opción más adecuada, ya que el sistema heredado alberga conocimiento y comprensión fundamental de la organización, y volver a implementarlo sería costoso y arriesgado. Evolucionar una arquitectura conlleva múltiples incertidumbres y desafíos, tal como describe (Richardson, 2018). Entre las cuestiones más notables se incluyen la restricción de recursos, la necesidad de mantener la operatividad de forma constante y el riesgo asociado en la incorporación de nuevas tecnologías (Barnes et al., 2012).

2.2.4. Descomposición arquitectónica

Es importante tener en cuenta que la arquitectura de software se refiere a la organización estructurada de elementos en los sistemas de software. Sin embargo, existe un proceso inverso en el cual cada uno de los componentes que conforman el sistema se divide, manteniendo la integridad de la información o el contenido original del sistema (Lung et al., 2007). En la literatura, este proceso de separación se denomina comúnmente "*descomposición arquitectónica*".

La descomposición arquitectónica engloba todas las estrategias, prácticas o modelos que posibilitan la subdivisión de sistemas de software en componentes más pequeños, con el propósito de mejorar los procesos de construcción, desarrollo, integración, comunicación y evolución (Orjuela Velandia, 2022).

Por lo general, la disposición de estas estructuras se compone de un conjunto de entidades y relaciones que mantienen una asociación sólida entre sí. Además, cuando los componentes o elementos que se dividen tienen un proceso de comunicación bien definido y están desglosados, de manera que existe una coherencia global en el software, la integración de estos conforma y constituye a un sistema de software compuesto (Mitchell & Mancoridis, 2006).

2.2.5. Arquitecturas monolíticas

Los sistemas de software tradicionales en la arquitectura monolítica constan de tres componentes fundamentales. La primera capa es la interfaz de usuario, que opera principalmente en el lado del cliente, mientras que el servidor suministra recursos que suelen estar en formato HTML, JS o CSS. La segunda capa es la lógica empresarial, es la lógica del negocio encargada de interpretar las peticiones de los clientes y generar respuesta a estas. Por último, se encuentra la

tercera capa, encargada de gestionar la persistencia de la información (Richardson, 2014).

Las principales características distintivas de los monolitos son las siguientes:

a. **Única unidad de despliegue:** Los monolitos se componen de un conjunto de módulos que se ejecutan de manera integrada, esto implica que la mayoría de las funciones del sistema residen en un único proceso de ejecución (Newman, 2019).
b. **Comunicación a través de memoria:** Esta característica deriva de la propiedad previa y consiste en la habilidad inherente de los monolitos de comunicar información entre sus diferentes módulos y capas a través de la memoria, debido a que una de sus características principales de sus procesos implica la transferencia de datos a través del medio en cuestión (Richardson, 2014).
c. **Dependencia tecnológica:** Los monolitos son desarrollados e implementados en un Stack tecnológico de propósito general, el cual está vinculado con el entorno de ejecución, lo que hace inviable incorporar más de un Stack en el componente. Como consecuencia, la arquitectura a menudo no solo crea una dependencia en la tecnología, sino también en una versión particular de la misma (Newman, 2019).
d. **Repositorio de control de versiones unificado:** Como lo explica, a pesar de que el enfoque monolítico no exige necesariamente que todo el código base de la aplicación se encuentre completamente unificado, en la mayoría de las aplicaciones que siguen esta arquitectura es una práctica común que efectivamente esté centralizado, lo cual implica que la gestión de versiones se concentre en un único repositorio (Newman, 2019).

Estas propiedades confieren a las arquitecturas monolíticas una serie de ventajas y desventajas. A continuación, se tiene algunas de las ventajas que ofrece el estilo arquitectónico monolítico:

- **Topología simple de despliegue:** Para lograr un sistema operativo funcional es necesario ejecutar los componentes internos del sistema. Sin embargo, debido a la característica **(a)**, el número de componentes suele ser reducido, ya que el monolito contiene la mayor parte de la funcionalidad. Esto facilita a que los despliegues sean mucho más sencillos desde su concepción. Ya que, en la mayoría de los casos, actualizar el sistema o reiniciar la compilación implica detener el proceso principal (Newman, 2019).
- **Monitoreo del sistema:** Este beneficio también, por la característica **(a)**, ya que supervisar un sistema se simplifica al observar el reducido conjunto de componentes del sistema. Esto representa una ventaja significativa, sobre todo cuando se cuenta con equipos pequeños o recursos limitados, ya que implica menos esfuerzo y no es necesario contar con una alta tecnificación para llevar a cabo la supervisión (Newman, 2019).
- **Implementación de operaciones en múltiples dominios:** En muchas ocasiones, es necesario fusionar información proveniente de diversos sectores de la aplicación con el fin de proporcionar una respuesta integral al usuario. La característica **(b)**, en aplicaciones monolíticas, permite llevar a cabo esta tarea mediante el uso de memoria, lo que conlleva ventajas en términos de rendimiento, simplificación en la implementación, reducción en el tiempo necesario para llevar a cabo esta operación, y delegar el manejo de errores al sistema operativo, que se encarga de todos los aspectos relacionados con el control del proceso (Newman, 2019).

- **Reutilización de código:** Una práctica efectiva en el desarrollo de software es la codificación de funcionalidades con una alta cohesión, lo que representa varias ventajas, incluyendo la posibilidad de reutilizar código basado en la funcionalidad. Gracias a las características **(c y d)** este proceso es viable en arquitecturas monolíticas, donde a menudo se generan módulos con funcionalidades genéricas o comunes al sistema. Esto conlleva ahorros de tiempo en el desarrollo y mantenimiento, así como la reducción de posibles errores durante la implementación. No obstante, es esencial tener en cuenta que esta práctica debe utilizarse con precaución, ya que puede incrementar la dependencia en el acoplamiento del sistema (Gamma, 2002).
- **Implementación de pruebas de software:** Básicamente, la elaboración de pruebas de software para un componente arquitectónico que concentra la mayor parte de la funcionalidad del sistema resulta más sencillo que hacerlo para otras arquitecturas donde las funciones fundamentales se distribuyen en diferentes componentes, como en el caso SOA (Service Oriented Architecture) o Microservicios, sobre todo en lo que respecta a cuestiones de integración (Newman, 2019).

No obstante, las arquitecturas monolíticas presentan inconvenientes que han impulsado a lo largo del tiempo la evolución desde estas estructuras tradicionales hacia arquitecturas contemporáneas. Algunas de estas desventajas incluyen:

- **Escalamiento horizontal:** Es frecuente que los elementos monolíticos realicen un alto consumo de recursos de infraestructura en su ambiente productivo. Una implicación de la característica **(a)** es que cuando la demanda del sistema aumenta y se opta por aplicar un escalado horizontal como

solución a dicha demanda, se genera la necesidad de crear réplicas adicionales del sistema que consumirán recurso en una proporción semejante. Esto conlleva un incremento en los gastos de infraestructura y vuelve insostenible mantener a largo plazo el modelo de escalamiento (Newman, 2019). Además, si consideramos que la mayoría de las situaciones, el aumento en la demanda se debe a un servicio particular y no se requiere escalar todo el componente, esto representa una desventaja significativa para este enfoque.

- **Despliegue de cambios pequeños:** A veces, es necesario implementar ajustes menores en un entorno de producción, esto está implicada por a la característica **(a)** la cual requiere que se lleve a cabo todo el proceso de despliegue. De acuerdo con el proceso establecido, esto puede implicar que algunos sistemas deben estar fuera de servicio durante un determinado periodo, lo que obstaculiza la posibilidad de realizar despliegues continuos.
- **Vulnerabilidad al alto acoplamiento:** Las arquitecturas monolíticas no son causa ni consecuencia de un acoplamiento elevado, pero pueden facilitar su desarrollo. Si los desarrolladores que trabajan en la implementación de la arquitectura no siguen prácticas estrictas en el diseño y la ejecución, es común que se creen conexiones perjudiciales en el sistema, lo que conlleva un alto grado de acoplamiento (Newman, 2019). En cierto sentido, esto representa un efecto negativo de la reutilización de código, que es una de las ventajas destacadas de esta arquitectura.
- **Actualización tecnológica:** Conforme se explica en la característica **(c)**, una aplicación monolítica se encuentra limitada a utilizar un único conjunto de tecnologías, lo que crea una sólida dependencia de la aplicación respecto a la tecnología. Esto dificulta la realización de actualizaciones tecnológicas

frecuentes en la aplicación, ya que existe el riesgo de que las funciones utilizadas cambien su comportamiento habitual y provoquen errores. Normalmente, llevar a cabo ajustes en cambios menores de versión resulta relativamente sencillo, pero realizar cambios significativos de versión conlleva una exhaustiva revisión y una serie de pruebas para garantizar el correcto funcionamiento del software (Richardson, 2014).

- **Condicionada a limitaciones tecnológicas:** Como es de esperar, cada tecnología presenta una serie de ventajas y desventajas. Debido a la característica **(c)**. Una arquitectura monolítica se ve afectada y condicionada por las limitaciones de la tecnología.
- **Mayor probabilidad de conflictos en el control de versiones:** Cuando se necesita la colaboración de múltiples desarrolladores para implementar o dar soporte a un monolito, aumenta la posibilidad de que dos o más desarrolladores choquen al realizar cambios en los mismos archivos. Esto crea conflictos en el sistema de control de versiones, especialmente cuando se cumple con la característica **(d)**. La probabilidad de conflictos se incrementa a medida que se incrementa el número de personas que deben interactuar con el monolito.

2.2.6. Arquitectura de microservicios

Dentro de esta fase de investigación, otro concepto fundamental es la arquitectura de Microservicios, que se basa en la distribución funcional del software a través de unidades más pequeñas conocidas como Microservicios. Estos Microservicios están diseñados con un conjunto de características y propiedades específicas orientadas a abordar problemas particulares dentro de un dominio concreto (Newman, 2019).

Esta evolución tiene por objetivo la creación de múltiples puntos de procesamiento de datos de manera autónoma, sin sacrificar la integridad del sistema y manteniendo el principio de independencia. En estas arquitecturas, se busca mantener el enfoque en la distribución de servicios y, al mismo tiempo, se procura que el conjunto de servicios más pequeños cumpla con los estándares definidos en las decisiones de diseño. Esto garantiza que la estructura del sistema funcione de manera efectiva y cumpla con los requisitos del modelo (Newman, 2015).

Las arquitecturas de Microservicios surgen como una opción para abordar los desafíos frecuentes que se presentan en estilos como las arquitecturas monolíticas (Richardson & Smith, 2016). Este enfoque se fundamenta en la computación distribuida y consta de un conjunto de elementos enfocados en servicios de pequeña escala que están diseñados en función de las necesidades del negocio. Estos servicios operan de manera autónoma e independiente, y su interacción se lleva a cabo de manera coordinada a través de protocolos ligeros (Newman, 2015).

Volviendo a la disposición convencional de capas en el estilo cliente-servidor, que consta de tres módulos principales, en una arquitectura de Microservicios, sobre todo los módulos de lógica de negocio y datos se fragmentan en servicios más pequeños, limitando así el alcance de su dominio. Esto no implica que el módulo de interfaz de usuario no pueda dividirse en Microservicios, aunque esta división es menos común (Newman, 2019).

Un desafío significativo en la arquitectura radica en establecer los límites de cada Microservicio. La fragmentación del software en servicios minúsculos,

generalmente engloban un dominio específico o un conjunto de funcionalidades diseñadas para un propósito singular. En este punto, vale la pena mencionar que la noción de Microservicio viene de la idea semántica de un servicio de software, pero se centra en una entidad extremadamente pequeña con límites bien definidos debido a su dimensión reducida (Rivera & Humberto, 2021). Dado que las arquitecturas de Microservicios tienden a ser complejas, es común aplicar varios patrones que contribuyen a simplificar el funcionamiento del sistema (Lewis & Fowler, 2014).

Cada uno de los componentes es independiente en todas las etapas de su desarrollo, que incluyen análisis, diseño, implementación, pruebas y mantenimiento. No obstante, es esencial que operen de manera coordinada y conjunta para satisfacer los requisitos del sistema en su conjunto (Christopher Newman et al., 2018).

A continuación, se enumeran algunas de las características más destacadas de los Microservicios:

a. **Servicios pequeños con procesos independientes:** La característica central de los Microservicios radica en fragmentación de una aplicación en varios módulos de manera modular, donde cada módulo desempeña una función específica y restringida en el sistema. Según su definición precisa, cada uno de estos módulos ejecuta un proceso que es completamente autónomo con relación a otros módulos (Newman, 2015).
b. **Arquitectura distribuida:** Dado que esta arquitectura comprende varios componentes, en muchas situaciones es necesario consultar dos o más de estos componentes para satisfacer una solicitud. Cada componente proporcionará la

parte de información relacionada con su área de competencia, y de esta manera se recopila la información completa para generar la respuesta deseada (Newman, 2019).

c. **Comunicación mediante protocolos ligeros:** Dado que se trata de una arquitectura distribuida con procesos independientes, no es factible la comunicación entre Microservicios a través de la memoria. En su lugar, la interacción entre los componentes debe llevarse a cabo mediante medios de red que posibiliten una transferencia eficiente de datos. Entre los enfoques más habituales, se incluyen REST y gRPC, que emplean el protocolo HTTP (Richardson, 2018).

d. **Base de datos independiente:** Una característica que deriva de los servicios independientes van definidos según un contexto o dominio, por ello es importante contar con bases de datos junto con sus sistemas de gestión que operan de manera independiente y representan las entidades que están dentro de los límites establecidos para el servicio (Harms et al., 2017).

e. **Autonomía:** Esta característica se basa en la idea de que cada una de estas piezas de software es una entidad independiente, con poca o ninguna relación con los demás componentes del sistema. Esto permite que el desarrollo de cada servicio pueda llevarse a cabo de forma independiente del resto de elementos del sistema, utilizando únicamente la información específica del dominio y las reglas de implementación establecidas en las decisiones de diseño (Newman, 2015).

f. **Modularidad:** Este principio es esencial en el patrón, ya que establece que los sistemas deben descomponerse en módulos para que puedan ser desarrollados, comprendidos y liberados por diferentes individuos. La noción

subyacente respecto a la modularidad implica que se especifiquen pequeños módulos donde se condense la funcionalidad general del sistema (Richardson, 2018).

g. **Paradigma de distribución:** Una característica inherente a este modelo implica que las arquitecturas deben buscar la descentralización, evitando así la presencia de puntos únicos de falla o dependencias significativas en un solo componente. El pilar de este patrón consiste en intentar desacoplar los dominios del sistema de la manera más granular posible, con el propósito de crear entidades independientes que sean más fáciles de analizar (Newman, 2015).

h. **Comunicación independiente:** Algunas de las características previamente mencionadas permiten que los Microservicios tengan protocolos de comunicación completamente independientes del resto de los componentes del sistema (Newman, 2015).

Claramente, la implementación de este patrón en un modelo, ya sea nuevo o existente, debe ser considerada durante la fase de análisis y diseño del desarrollo de software. Esto se debe a que no todos los entornos digitales son adecuados para migrar hacia sistemas distribuidos como la arquitectura de Microservicios. No obstante, la implementación de este tipo de arquitecturas en un sistema de software conlleva una serie de beneficios significativos que pueden ser aprovechados según el tipo y la finalidad del sistema. Por lo tanto, a continuación, se mencionan numerosas ventajas que ofrece esta arquitectura, algunas de las más notables incluyen:

- **Pluralidad tecnológica:** Debido a la característica **(a)**, cada uno de los Microservicios definidos tiene la capacidad de operar de manera autónoma y

sin depender de otros. Como resultado, se vuelve factible que cada Microservicio esté construido utilizando la tecnología más adecuada para su propósito, lo que facilita la posibilidad de contar con una arquitectura que abarque diversas tecnologías (Richardson, 2018).

- **Reducción de la complejidad:** La presencia de elementos pequeños con comportamientos previsibles simplifica la comprensión lógica y funcional de estas partes de software. Además, su tamaño reducido ofrece la oportunidad de eliminar posibles interdependencias informáticas durante las migraciones o modificaciones en el código, facilitando la implementación de métodos de comunicación sencillos y mejorando la interacción con otros componentes del sistema (Richards, 2022).
- **Actualización tecnológica sencilla:** Gracias a la característica **(a)**, los Microservicios dentro de la arquitectura están claramente delimitados en cuanto a su ámbito de dominio y funcionalidad. Cuando surgen actualizaciones tecnológicas, ya sean de envergadura menor o mayor, que requieren adaptación, el proceso se vuelve más rápido de llevar a cabo y de poner a prueba, ya que solo se considera lo que abarca el Microservicio, cuyo alcance es limitado (Richardson, 2018).
- **Adopción de nuevas tecnologías:** Esta ventaja contribuye a la diversidad y pluralidad tecnológica que proporcionan los Microservicios. Simplemente, se trata de desarrollar un nuevo Microservicio que proporcione la funcionalidad necesaria o reconstruir un Microservicio en la tecnología deseada, con el propósito de aprovechar al máximo las capacidades de esa tecnología (Richardson, 2018).

Dado que cada una de las unidades del servicio es una entidad independiente, existe la opción de desarrollar cada uno de estos elementos de forma aislada utilizando tecnologías independientes. Por lo tanto, la disponibilidad de una tecnología específica para cada caso permite seleccionar libremente un lenguaje de programación, una herramienta, una biblioteca o un protocolo según la necesidad. Esta diversidad no se limita únicamente a la fase de desarrollo, sino que también puede ser relevante en el contexto de una posible migración o actualización. Esto no solo conlleva una ventaja pragmática que depende de las funciones del servicio, sino que también asegura el desacoplamiento de las dependencias que podrían presentar en los componentes monolíticos anteriores (Newman, 2015).

- **Mayor autonomía para grandes equipos de desarrollo:** Cuando el departamento de tecnología de una empresa necesita una cantidad significativa de desarrolladores, es común que se formen equipos pequeños con un número de miembros que varía entre cuatro y siete. Dentro de una arquitectura de Microservicios, cada equipo cuenta con responsabilidades claramente definidas, ya que su ámbito de responsabilidad está delimitado por el alcance del dominio del conjunto de Microservicios que les han sido asignados (Newman, 2019). Debido a la naturaleza modular del diseño y desarrollo de los Microservicios, el equipo tiene un control total y la capacidad de operar de manera autónoma sobre el conjunto de Microservicios que están bajo su responsabilidad, sin estar altamente dependiente de otros equipos.
- **Reducción del Time To Market:** Una de las principales ventajas de los Microservicios es su capacidad de implementación independiente, lo cual se manifiesta en la frecuencia con la que se lanzan cambios pequeños o nuevas

funcionalidades en producción. A diferencia de las arquitecturas centralizadas, como las aplicaciones monolíticas, donde a menudo se debe esperar hasta que un conglomerado de cambios esté listo, esto a fin de justificar la interrupción del sistema, en estas arquitecturas distribuidas se pueden realizar despliegues de manera independiente sin causar un fuerte impacto en la disponibilidad del sistema (Newman, 2019).

- **Acoplamiento limitado:** Debido a las delimitaciones establecidas por los Microservicios, la interdependencia también estará restringida a los límites específicos de cada Microservicio (Orjuela Velandia, 2022).
- **Evolución y extensibilidad:** Dado que la mayoría de las características de este sistema se originaron como soluciones a las limitaciones de otras arquitecturas, se integraron las mejores cualidades de otros patrones con el propósito de crear un modelo sólido y robusto. Esto se puede observar en los procesos simplificados de entrega continua, los flujos de desarrollo, la comprensión de componentes desacoplados y las unidades distribuidas de despliegue. Por lo tanto, la evolución y el mantenimiento de este tipo de implementaciones se agilizan gracias al tamaño inherente de los Microservicios (Richards, 2022).
- **Eficiencia en el escalamiento:** En todos los sistemas informáticos, hay módulos particulares que experimentan una carga más elevada que otros. La arquitectura de Microservicios permite escalonar los módulos que necesitan manejar una mayor carga de trabajo, lo que aumenta la eficiencia en el uso de la infraestructura disponible (Orjuela Velandia, 2022).

Bajo este enfoque, los servicios que necesiten una mayor capacidad de respuesta o una mejora en su rendimiento pueden ser escalados de manera

individual o conjunta, según sea necesario. Esto contrasta significativamente con las arquitecturas monolíticas, en las cuales es necesario escalar todo el sistema, incluso cuando algunos componentes no lo requieren. Además, la ventaja del escalado en un sistema distribuido radica en cómo se gestionan los costos y los mecanismos de aprovisionamiento, ya que se ajustan según la demanda de cada Microservicio. Esto no solo resulta en un ahorro de costos para el sistema en su conjunto, sino que también proporciona métricas de rendimiento que reflejan mejoras, retroalimentación o cambios (Newman, 2019).

- **Facilidad en el despliegue:** Mediante el empleo de Microservicios, la implementación, reversión o actualización de cualquier magnitud no plantea desafíos en la etapa de despliegue. Dado que cada componente es autónomo y se ejecuta de manera aislada en su entorno, el proceso de puesta en marcha se lleva a cabo de manera independiente, sin afectar a otros componentes ni causar efectos colaterales. Esta ventaja también contribuye a que el sistema responda de inmediato a los cambios y no requiera extensos períodos de tiempo en etapas innecesarias (Newman, 2015).
- **Resiliencia:** Una ventaja adicional de implementar esta arquitectura es la capacidad de recuperación "*resiliencia*" que se encuentra integrada en sus Microservicios. Esto significa que si surge algún problema técnico en una parte del sistema relacionado con el desarrollo o la ejecución (ya sea un servicio, un conector o un módulo), el sistema tiene la capacidad de recuperarse de dicho fallo sin afectar el funcionamiento de ninguna de las funciones del sistema. Este aspecto contrasta de gran relevancia en comparación con otros patrones, dado que, en estas estructuras, cuando ocurre un fallo, generalmente no representan un impacto en la totalidad del sistema,

evitando la falta de disponibilidad del servicio y la necesidad de largos períodos de mantenimiento(Newman, 2015).

Un efecto de la arquitectura de Microservicios es la eliminación de Puntos Únicos de Fallo “SPOFs”. Cuando un microservicio experimenta un fallo, esto resulta en una breve interrupción en el servicio que proporciona, pero los demás servicios continúan funcionando. Lo mismo ocurre con los despliegues, ya que al ser independientes los procesos de implementación de un Microservicio, podría conllevar una breve interrupción en ese servicio en particular, mientras que el resto del sistema seguirá operativo. Esto conlleva un aumento en la disponibilidad general del sistema (Harms et al., 2017).

- **Habilitan la interoperabilidad interna y con otros sistemas:** Como se hizo referencia en la característica **(c)**, los Microservicios dependen de métodos de comunicación basados en redes y protocolos de bajo peso para interactuar entre sí. Esta particularidad posibilita que, sin importar la tecnología empleada por cada microservicio, se pueda compartir información no solo internamente, sino también externamente con otros sistemas (Richardson, 2018).
- **Dominio de datos:** De acuerdo con la característica **(d)**, cada Microservicio establece su propio ámbito de responsabilidad del negocio. Esto significa que cada componente ejerce control exclusivamente sobre sus propios datos, lo que facilita la toma de decisiones tecnológicas más precisas. Por ejemplo, se puede optar por emplear una base de datos relacional o no relacional, determinar el nivel de carga que la base de datos debe manejar y evaluar si se necesitan tácticas arquitectónicas adicionales, entre otras posibilidades (Harms et al., 2017).

- **Pruebas autónomas:** La capacidad de realizar pruebas sobre componentes de software más pequeños mediante mecanismos de pruebas unitarias o funcionales simplifica el proceso. Estos elementos pueden ser probados de manera independiente, sin necesidad de ejecutar algún proceso interno en paralelo (Richards, 2022).

Aunque las arquitecturas de Microservicios ofrecen numerosas ventajas, es importante considerar algunas desventajas al tomar decisiones de este tipo:

- **Mayor complejidad en la administración y monitoreo del sistema:** Mientras que en las arquitecturas monolíticas solo se requiere implementar un componente, en las arquitecturas de Microservicios, debido a la característica **(a)**, es necesario gestionar todos los componentes por separado. Además, supervisar y mantener el estado adecuado de todos estos componentes es una tarea más complicada. Es evidente que, a medida que aumenta el número de Microservicios, la complejidad de su administración y seguimiento también se incrementa (Newman, 2015).
- **Aumento de la latencia en el sistema:** En todos los sistemas con el estilo cliente-servidor, la latencia está condicionada por la calidad de la red utilizada. En arquitecturas convencionales, como las monolíticas, el tiempo de respuesta ante una solicitud no solo depende de la velocidad de procesamiento de los componentes de la aplicación y la base de datos, sino también de la latencia del conector que se encuentra entre estos componentes. Muchas de las solicitudes realizadas en las arquitecturas de Microservicios requieren la participación de dos o más componentes para generar una respuesta completa, y debido a las características **(b y c)**, se generan inevitablemente demoras adicionales en el sistema (Richardson, 2018).

- **Dependencia del canal de comunicación:** Otra de las principales desventajas de estas arquitecturas es que se vuelven altamente dependientes del canal de comunicación. Como se ha mencionado en repetidas ocasiones, en muchas solicitudes, los Microservicios necesitan interactuar con otros Microservicios, y esta interacción solo es posible a través de la red. Además, si la red se congestiona, la latencia aumenta y los servicios tardarán más tiempo en finalizar la operación deseada (Richardson, 2018).
- **Alto costo en los errores de diseño:** Los fallos en el diseño de las arquitecturas distribuidas resultan más costosas en comparación con las arquitecturas tradicionales. A pesar de que se ha mencionado que la adopción de nuevas tecnologías y actualizaciones es más sencilla en esta arquitectura, el desarrollo completo y las modificaciones en los límites de un Microservicio implican un mayor costo en términos de tiempo y esfuerzo en comparación con una arquitectura monolítica (Newman, 2019). Esto se debe a la necesidad de configurar las interacciones con otros componentes de acuerdo con lo especificado en la característica **(c)**.
- **Complejidad para las transacciones en el sistema:** Cuando una transacción involucra solo a un Microservicio, la complejidad puede ser similar a la de una aplicación monolítica. Sin embargo, en situaciones en las que las transacciones abarcan más de un Microservicio, el proceso se vuelve más complicado. Una consecuencia de la característica **(d)**. Es la pérdida de la integridad referencial que normalmente proporcionan los motores de bases de datos, lo que implica trasladar la responsabilidad a los propios Microservicios o a una capa intermedia que ofrezca esa funcionalidad (Harms et al., 2017).

Ambas soluciones planteadas requieren un esfuerzo significativo y aumentan la complejidad del sistema.

2.2.7. Calidad de software

La calidad del software se describe como un proceso eficiente de desarrollo de software que se aplica de manera que produce un producto valioso y útil, ofreciendo un valor medible tanto para quienes lo desarrollan como para quienes lo utilizan (Pressman, 2010). Por otra parte, Kniberg (2015) distingue entre la calidad interna y externa de la siguiente manera:

- **Calidad externa:** Se refiere a lo que los usuarios del sistema experimentan y perciben. Un ejemplo de baja calidad externa sería una interfaz de usuario lenta y poco intuitiva.
- **Calidad interna:** Hace referencia a los aspectos que, por lo general, no son evidentes para el usuario, pero que tienen un impacto significativo en la capacidad de mantener el sistema. Esto incluye aspectos como la coherencia en el diseño del sistema, la amplitud de cobertura de las pruebas, la legibilidad del código, la refactorización, entre otros.

En términos generales, es posible que un sistema con una alta calidad interna aún presente una baja calidad externa, pero un sistema con baja calidad interna rara vez logrará tener una buena calidad externa. En contraste, la ISO/IEC 25000 es una serie de normas que se propone establecer un marco de trabajo común para la evaluación de la calidad de un producto de software (*Portal ISO/IEC 25000*, 2023).

2.2.7.1. La familia de normas ISO/IEC 25000

ISO/IEC 25000, también conocida como SQuaRE (System and Software Quality Requirements and Evaluation), constituye una familia de estándares diseñada para establecer un marco unificado con el propósito de evaluar la calidad de los productos de software. Esta familia de normas, la ISO/IEC 25000, es el resultado de una evolución a partir de normativas previas, principalmente de las normas ISO/IEC 9126, que definen un modelo específico para la calidad de productos de software, así como la ISO/IEC 14598, que se enfocaba en el proceso de evaluación de estos productos. La familia ISO/IEC 25000 abarca cinco divisiones distintas (*Portal ISO/IEC 25000*, 2023).

Figura 2

ISO/IEC 25000.

Fuente: (Portal ISO/IEC 25000, 2023)

2.2.7.2. ISO/IEC 25010

Según la norma ISO 25010, el modelo de calidad sirve como el núcleo esencial que guía la estructuración del sistema para la evaluación de la calidad del producto. En este modelo se detallan las características de calidad que se tendrán

en cuenta al evaluar las propiedades de un software particular (*Portal ISO/IEC 25000*, 2023).

La calidad del software puede entenderse como el nivel en que el producto cumple con los requisitos de sus usuarios, proporcionando así un valor. Estos requisitos, que incluyen funcionalidad, rendimiento, seguridad, mantenibilidad, entre otros, están reflejados en el modelo de calidad. Este modelo organiza la calidad del producto en diversas características y sub características (*Portal ISO/IEC 25000*, 2023).

El modelo de calidad del producto, definido por la norma ISO/IEC 25010, se fundamenta en ocho atributos de calidad que están delineados en Figura 3.

Figura 3

ISO/IEC 25010 Características de calidad.

Fuente: (Portal ISO/IEC 25000, 2023)

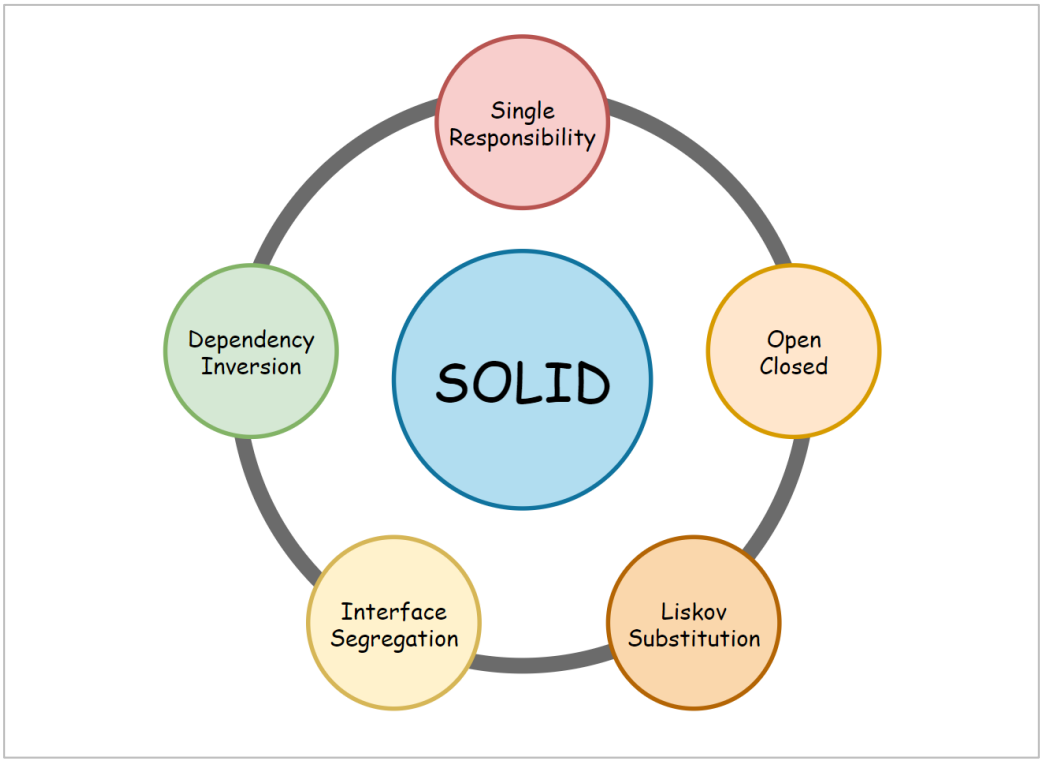
2.2.8. Principios SOLID

La sigla "*SOLID*" representa un conjunto de principios y buenas prácticas que, al ser aplicados e implementados de manera integral, facilitan la capacidad del código para adaptarse a cambios futuros. Estos principios SOLID fueron introducidos por Bob Martin hace casi 15 años. Sin embargo, a pesar de su

utilidad, estas prácticas no son tan ampliamente reconocidas como podrían o deberían serlo (Hall, 2014). La Figura 4 presenta los cinco fundamentos que componen el conjunto SOLID.

Figura 4

Principios de diseño S.O.L.I.D.



Fuente: (Martin & Kriens, 2012)

En la Tabla 1, se presenta una concisa explicación de cada uno de los principios SOLID. Estos principios se centran en aspectos particulares de la programación orientada a objetos, contribuyendo así a la creación de código más limpio y fácil de entender.

Tabla 1

Principios S.O.L.I.D.

SRP	**El Principio de Responsabilidad Única**	Una clase debe tener una, solo una razón para cambiar.
OCP	**El Principio Abierto/Cerrado**	Debe ser capaz de extender un comportamiento de clases, sin modificarlo.
LSP	**El Principio de Sustitución de Liskov**	Las clases derivadas deben ser sustituibles por sus clases base.
ISP	**El Principio de Segregación de Interfaces**	Hacer interfaces de grano fino que son específicos del cliente.
DIP	**El Principio de Inversión de Dependencia**	Depende de abstracciones, no de concreciones.

Fuente: (Martin & Kriens, 2012)

2.2.8.1. Principio de responsabilidad única

> **Principio de Responsabilidad Única (SRP)**
>
> *Una Clase debe tener solo una razón para cambiar.*
>
> (Martin & Martin, 2006)

El principio de responsabilidad única orienta a los desarrolladores a crear código que posea una única razón para cambiar. Si una clase tiene más de una razón para cambiar, implica que asume más de una responsabilidad. En tales casos, se debe dividir la clase en múltiples clases más pequeñas, cada una con una sola responsabilidad y una razón única para cambiar (Hall, 2014). Este principio

nos proporciona una definición de responsabilidad y establece una pauta para el tamaño de las clases. Todos los sistemas contienen una lógica y complejidad significativas. El propósito principal de gestionar esta complejidad es estructurarla de manera que un programador sepa dónde buscar y comprenda la complejidad que está directamente relacionada en cada momento específico (Martin & Kriens, 2012).

En cambio, un sistema que utiliza clases más grandes con múltiples responsabilidades nos obliga a buscar entre numerosos elementos que no siempre son relevantes o necesarios para nuestra tarea. Hace hincapié en la idea de que los sistemas deben estar compuestos por muchas clases pequeñas en lugar de unas pocas clases de gran tamaño. Cada clase pequeña debe encapsular una única responsabilidad, tener una sola razón para cambiar y colaborar con otras para lograr los comportamientos deseados del sistema (Martin, 2009). La Figura 5 muestra un ejemplo de una clase que sigue el principio de responsabilidad única.

Figura 5

Una clase con una única responsabilidad.

```
public class Version {
        public int getMajorVersionNumber();
        public int getMinorVersionNumber();
        public int getBuildNumber();
}
```

Fuente: (Martin, 2009)

2.2.8.2. Principio abierto/cerrado

Principio de Abierto y Cerrado (OCP)

Las entidades de software (Clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación.

(Martin & Martin, 2006)

Según Hall (2014) en su libro "*Adaptive code via C# : Agile coding with design patterns and SOLID principles. Microsoft Press*". Existen dos formulaciones del principio abierto/cerrado que requieren ser analizadas.

- **Según la definición de Meyer:** Las entidades de software deben permitir la extensión, pero no deben permitir modificaciones.
- **Según la definición de Martin:**
    - **Abierto para extensión:** Implica que el módulo puede ser ampliado para incluir nuevos comportamientos a medida que cambian los requisitos de la aplicación. En otras palabras, es posible cambiar lo que hace el módulo.
    - **Cerrado para modificación:** Significa que la extensión del comportamiento del módulo no conlleva cambios en el código fuente o binario del módulo.

2.2.8.3. Principio de sustitución de liskov

> **Principio de Sustitución de Liskov (LSP)**
>
> *Los subtipos deben ser sustituibles por sus tipos base.*
>
> (Martin & Martin, 2006)

Barbara Liskov formuló este principio en 1988 y expresó: "Lo que estamos buscando aquí es algo similar a la siguiente propiedad de reemplazo: Si, para cualquier objeto ***O1*** de tipo ***S***, existe un objeto ***O2*** de tipo ***T*** de manera que, para todos los programas ***P*** definidos en términos de ***T***, el comportamiento de ***P*** no se

altera cuando se reemplaza ***O1*** por ***O2***, entonces se puede afirmar que ***S*** es un subtipo de ***T***" (Martin & Martin, 2006).

La importancia de este principio se hace evidente cuando se consideran las consecuencias de su incumplimiento. Imaginemos que tenemos una función ***F*** que toma como argumento una referencia a una clase base ***B***. Ahora supongamos que cuando pasamos ***F*** en la forma de ***B***, algún derivado ***D*** de ***B*** hace que ***F*** funcione de manera incorrecta. En este caso, ***D*** estaría violando el Principio de Sustitución de Liskov. Es evidente que ***D*** sería frágil cuando se trata de ***F***. Los desarrolladores de ***F*** podrían verse tentados a introducir algún tipo de verificación especial para ***D***, con el fin de que ***F*** pueda funcionar correctamente cuando se le pasa un objeto de tipo ***D***. Estas pruebas infringen el Principio Abierto/Cerrado debido a que ***F*** ya no es inmutable y adaptable a todos los posibles derivados de ***B*** (Martin & Martin, 2006).

2.2.8.4. Principio de segregación de interfaces

> **Principio de Segregación de Interfaces (ISP)**
>
> *Los clientes no deben verse obligados a depender de métodos que no utilizan.*
>
> (Martin & Martin, 2006)

Este principio se centra en abordar los inconvenientes de las interfaces "*voluminosas*". Las clases que presentan interfaces poco cohesionadas se consideran que tienen interfaces "*voluminosas*". En otras palabras, las interfaces de la clase pueden dividirse en grupos de métodos, y cada grupo se destina a atender a un conjunto distinto de clientes.

Por lo tanto, algunos clientes hacen uso de un conjunto de métodos y otros clientes utilizan los otros conjuntos. El Principio de Segregación de Interfaces

reconoce que hay objetos que necesitan interfaces no cohesivas. No obstante, sugiere que los clientes no deben estar al tanto de estas interfaces como una única clase. En su lugar, los clientes deben conocer clases base abstractas que tengan interfaces cohesivas (Martin & Martin, 2006).

Cuando los clientes se ven forzados a depender de métodos que no utilizan, quedan expuestos a posibles cambios en esos métodos. Esto conduce a un acoplamiento no deseado entre todos los clientes. En otras palabras, cuando un cliente depende de una clase que incluye métodos que no utiliza, pero que son necesarios para otros clientes, ese cliente se verá impactado por los cambios que esos otros clientes puedan introducir en la clase. Nuestra intención es evitar este tipo de acoplamientos siempre que sea posible, y por eso buscamos separar las interfaces (Martin & Martin, 2006).

2.2.8.5. Principio de inversión de dependencias

Principio de Inversión de Dependencias (DIP)

A. *Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.*
B. *Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.*

(Martin & Martin, 2006)

La aplicación del Principio de Inversión de Dependencias da lugar a la técnica bien conocida de inyección de dependencias, que se considera una de las mejores prácticas para gestionar las interacciones entre clases. Esta técnica genera código que es reutilizable, elegante y capaz de adaptarse a cambios sin desencadenar consecuencias en cascada. Establece que un módulo específico ***A*** no debe depender directamente de otro módulo concreto ***B***, sino que debe

depender de una abstracción de ***B***. Esta abstracción puede ser una interfaz o una clase abstracta que sirve como base para un conjunto de clases derivadas (Martin & Martin, 2006).

2.2.9. Clean architecture

Es un patrón arquitectural de diseño ágil de software propuesto por Robert C. Martin, autor de la reconocida serie de principios de diseño de software llamada SOLID, propuso un patrón arquitectural ágil para el desarrollo de software. Este patrón implica la organización del código en capas adyacentes, donde cada capa solo puede comunicarse con las capas ubicadas a sus lados. El propósito es abordar una serie de desafíos y mejorar la solidez, mantenibilidad, capacidad de pruebas y capacidad de extensión del código. Aunque existan diferencias en los detalles, todas estas arquitecturas comparten similitudes significativas. Su enfoque principal es la separación de responsabilidades en el software, lograda mediante la división del software en capas. Es común contar con al menos una capa para las reglas de negocio y otra para las interfaces de usuario y del sistema (Martin, 2017).

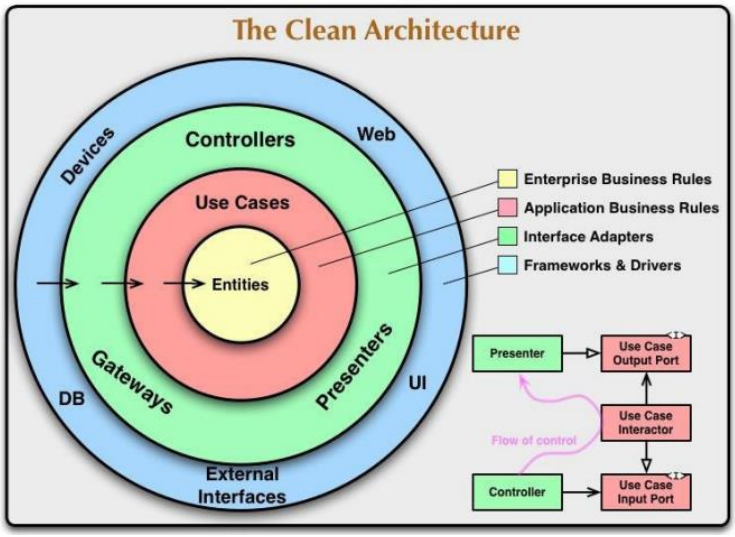
La regla fundamental que rige esta arquitectura es la regla de dependencias, que establece que las dependencias en el código fuente deben dirigirse exclusivamente hacia el interior. En otras palabras, lo que una capa externa pueda contener no será visible para una capa interna. Con el objetivo de preservar la integridad de las capas internas, se desaconseja el uso de estructuras de datos definidas en las capas externas en dichas capas. Únicamente se permite el paso de datos con una estructura simple de una capa a otra (Martin, 2017).

Las componentes fundamentales que constituyen la Clean Architecture en este caso de tipo Onion son Framework & Drivers, Interface Adapters,

Application Business Rules y Enterprise Business Rules. Estas capas están dispuestas en orden, desde la capa más externa hasta la más interna (Martin, 2017).

Figura 6

Diagrama Integrado de Onion Architecture.



Fuente: (Martin, 2017)

- La capa **Frameworks & Drivers** se constituye mediante las herramientas y Frameworks, como la base de datos o los Frameworks web, que se emplearán en la aplicación.
- La capa **Interface Adapters** tiene la responsabilidad de convertir los datos en un formato comprensible para la próxima capa que los utilizará. Además, albergará los controladores, presentadores y accesos a servicios de terceros.
- La capa de **Application Business Rules** engloba los casos de uso y, consecuentemente, la lógica de la aplicación. En este nivel se especificarán los datos de entrada, los datos de salida y el comportamiento del sistema.
- La capa final es **Enterprise Business Rules,** donde se alojan las entidades que incorporan las reglas de negocio. Estas entidades deben estar formadas por funciones fundamentales y tienen la particularidad de poder ser utilizadas por

uno o más componentes, lo que las hace independientes y exentas de cambios influenciados por elementos externos.

Habrá situaciones en las que una capa interna necesitará datos de una capa externa, pero, como se mencionó anteriormente, una capa interna no puede acceder directamente a una capa externa debido a la regla de dependencias. Para abordar esto, se emplea el principio de inversión de dependencias, un principio dentro de los principios SOLID. Este principio se fundamenta en dos reglas: la primera establece que la capa externa no debe depender de la capa interna, y ambas capas deben depender de abstracciones. La segunda regla dicta que las abstracciones no deben depender de detalles, sino que los detalles deben depender de las abstracciones (Martin, 2017).

En términos generales, cada una de las arquitecturas suele producir sistemas que presentan las siguientes características:

- **Independencia de Frameworks:** La arquitectura no depende de la existencia de una biblioteca de funciones, lo que implica que puede utilizar los marcos como herramientas y no estar restringida por sus limitaciones.
- **Testeable:** Las reglas de negocio pueden evaluarse en la interfaz de usuario, la base de datos, el servidor web u otros componentes externos.
- **Independiente de la interface de usuario:** La interfaz puede ser cambiada fácilmente sin necesidad de realizar modificaciones en el resto del sistema.
- **Independiente de la Base de Datos:** Es posible sustituir Oracle o SQL Server con Mongo, BigTable, CouchDB u otro sistema de base de datos, ya que las reglas de negocio no están atadas a una base de datos en particular.

- **Independiente a cualquier agente externo:** Las reglas de negocio no poseen conocimiento de las interfaces que conectan con el entorno externo.

2.2.10. Cohesión

La Alta Cohesión es uno de los principios fundamentales en la ingeniería de software que se refiere a la organización de las funciones con intereses comunes dentro de un componente de software, asegurando que estén en el mismo dominio y tengan piezas con responsabilidades exclusivas. Además, este principio se utiliza para evaluar en qué medida un módulo de software cumple con los objetivos establecidos en la fase de diseño (Jha et al., 2014).

Por lo tanto, la presencia de una alta cohesión en una arquitectura resulta beneficioso para el sistema, ya que garantiza que todas las características relacionadas se encuentren agrupadas, lo que facilita en gran medida los procesos de modificación, actualización o eliminación de funciones (Aral & Ovatman, 2013).

Además, dado que el concepto de cohesión resalta la interacción funcional entre los elementos de un módulo de software, se pueden evaluar decisiones arquitectónicas relacionadas con el desacoplamiento. Un aspecto fundamental de esta propiedad es que la cohesión se basa en el principio de responsabilidad única, lo que asegura que las implementaciones altamente cohesivas siempre estarán claramente definidas debido a su único propósito de cambio (Hall, 2014).

Al igual que tener alta cohesión en un sistema, no solo simplifica el mantenimiento, la evolución y las pruebas, sino que también su independencia funcional contribuye a reducir la complejidad del sistema, ya que los dominios están adecuadamente separados y no dependen entre sí (Saadati & Motameni,

2014). De esta manera, se puede comprender cómo la alta cohesión se relaciona con el patrón arquitectónico de Microservicios, ya que el desacoplamiento de las diversas unidades de servicio debe basarse en una adecuada separación según el dominio y los requisitos del sistema (Newman, 2015).

A continuación, se presentan algunas de las características que deben considerarse en la arquitectura de Microservicios con respecto a la cohesión:

a. Los Microservicios son entidades de servicio autónomas que poseen una única responsabilidad y operan en conjunto para cumplir un único propósito. Su función principal es llevar a cabo una característica específica en el entorno de software, sin exceder los límites de implementación definidos en las decisiones de diseño (Richardson, 2018).

b. La posibilidad de que un grupo de trabajo independiente pueda desarrollar un Microservicio contribuye a una mayor cohesión y autonomía organizacional, ya que cada grupo se enfoca en una única funcionalidad sin depender de otros componentes estructurales (Newman, 2019).

c. Cuando un Microservicio abarca dos características o dominios distintos, la mejor estrategia es desarrollar cada característica de manera independiente para evitar cualquier tipo de acoplamiento (Newman, 2015).

d. Para contribuir al tema del bajo acoplamiento en una pieza de software, se puede considerar la opción de que cada componente de servicio tenga su propia base de datos, aunque esto no sea una regla universal (Richardson, 2018).

e. La construcción de los Microservicios se ajusta a las restricciones del negocio, lo que implica que no deben realizar más ni menos de lo

necesario, ya que esto podría generar fragmentos de código innecesarios (Newman, 2019).

f. Un Microservicio no debe exponer toda su funcionalidad a diversas entidades en un sistema de software, a menos que sea un componente altamente interconectado, ya que esto puede llevar a la creación de dependencias y aumentar su grado de acoplamiento (Newman, 2015).

g. Los cambios en un Microservicio deben ser independientes para minimizar el acoplamiento y prevenir afectaciones externas, asegurando que las actualizaciones no impacten otras funcionalidades del sistema de software (Richardson, 2018).

2.2.11. Acoplamiento

El Bajo Acoplamiento es otro de los elementos clave que se pueden considerar como parte de los atributos de calidad del software. Este principio se vincula con la medida en que un componente de software depende de otro. Ciertamente, este concepto busca asegurar que el grado de dependencia entre cualquier elemento del sistema no tenga un impacto crítico en la estructura general, ya que mantener relaciones cercanas puede ocasionar complicaciones en los procesos de diseño y desarrollo del software (Orjuela Velandia, 2022).

Además, este principio se basa en la noción de que las partes del software deben tener una mínima o nula dependencia entre sí, lo que permite realizar cambios o modificaciones sin afectar las funciones de otros componentes del sistema (Harrison et al., 1998).

Por supuesto, la interacción y cooperación entre las diversas partes del sistema son cruciales para su funcionamiento, pero en este análisis también se

debe evaluar el nivel de complicación que resulta de relacionar una entidad con otra (Harrison et al., 1998).

Del mismo modo, el principio de bajo acoplamiento busca prevenir las repercusiones no deseadas de cualquier alteración en los componentes del sistema, asegurando que la relación entre las distintas características se base en la integración de datos en lugar de un intercambio directo de métodos o funciones (Saadati & Motameni, 2014). El enfoque de integración de datos generalmente se logra mediante el uso de una interfaz que actúa como un punto de acceso central para compartir información entre diferentes partes del software. Esto permite aislar las especificaciones internas de implementación de cada componente y, al mismo tiempo, mantener un acoplamiento mínimo en la medida de lo posible (Harrison et al., 1998).

CAPÍTULO III

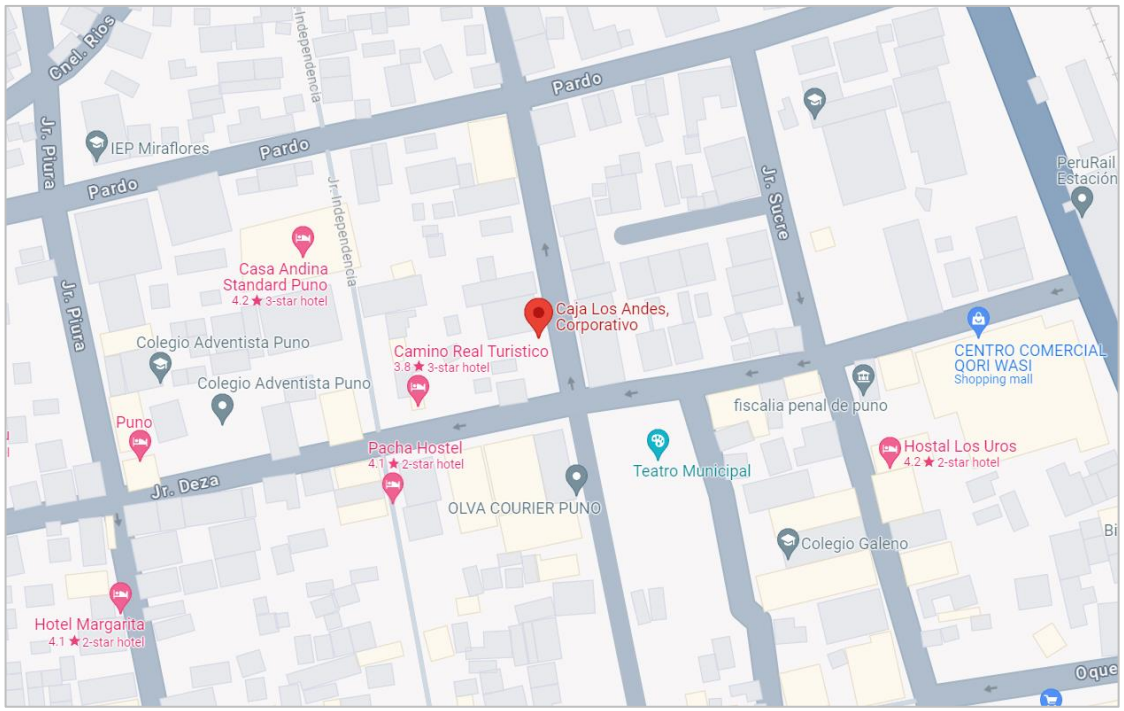
MATERIALES Y MÉTODOS

3.1. UBICACIÓN GEOGRÁFICA DE ESTUDIO

El proyecto se desarrolló en la ciudad de Puno e implemento en la Entidad Financiera los Andes de la misma ciudad de Puno - Jirón Junín N° 129, Se Muestra en la Figura 7. Esta institución fue creada oficialmente en noviembre de 1997. Es una institución orientada básicamente a la intermediación de las microfinanzas.

Figura 7

Ubicación geográfica de estudio.



Fuente: Google Maps.

3.2. OPERACIONALIZACIÓN DE VARIABLES

VARIABLE INDEPENDIENTE

X = Acoplamiento

VARIABLE DEPENDIENTE

Y = Cohesión

MATRIZ DE CONSISTENCIA

Tabla 2

Matriz de consistencia del proyecto.

	PREGUNTAS	OBJETIVOS	HIPÓTESIS	VARIABLES E INDICADORES	METODOLOGÍA
GENERAL	**Problema Principal** ¿Cuál estrategia de desacoplamiento aplicar en componentes Backend basados en Microservicios para evaluar la cohesión entre sus elementos, con el propósito de mejorar la eficiencia y escalabilidad del sistema?	**Objetivo Principal** Aplicar estrategias de desacoplamiento en componentes Backend basados en Microservicios, con el fin de evaluar la cohesión entre sus componentes.	**Hipótesis Principal** La aplicación de las estrategias de desacoplamiento en Microservicios Backend permite evaluar adecuadamente la cohesión entre sus componentes, en la entidad financiera Los Andes.	**VI:** Acoplamiento **VD:** Cohesión	**Enfoque** Cuantitativo **Diseño** Cuasi-Experimental **Tipo de Investigación** Aplicada **Técnicas e Instrumentos** - Se utilizará la Técnica de identificación de Microservicios usando descomposición funcional. - Se aplicará el patrón arquitectónico Onion Architecture basándose en los principios SOLID a fin de organizar y estructurar los servicios Backend. - Se evaluará empleando el concepto de árbol de subconjuntos planteado en el artículo científico titulado
ESPECÍFICAS	¿Cuáles son los elementos de desacoplamiento más relevantes en el patrón arquitectónico de Microservicios?	Identificar los elementos de desacoplamiento más significativos dentro del patrón arquitectónico de Microservicios utilizando descomposición funcional.	Al realizar un análisis utilizando descomposición funcional, se espera identificar los elementos de desacoplamiento más significativos, que mejoren la reducción del acoplamiento.		

¿Cómo diseñar y modelar una arquitectura de Backend basada en Microservicios, con el fin de lograr una arquitectura modular, escalable y eficiente?	Modelar una arquitectura de Backend basada en Microservicios, con el fin de lograr una arquitectura modular, escalable y eficiente.	Al modelar una arquitectura de Backend, se espera lograr una arquitectura altamente estructurada y escalable, con componentes independientes y acoplamiento reducido.	*"Measuring Cohesion And Coupling Of Object oriented Systems"*. **Muestra** Se tendrá como muestra los servicios de Backend del sistema Credirapp de la entidad financiera Los Andes.
¿Cómo evaluar la cohesión de los componentes resultantes del Backend, con el propósito de asegurar una arquitectura de alta calidad y eficiencia?	Evaluar la cohesión de los componentes resultantes del Backend mediante la aplicación de los principios de cohesión en Microservicios.	Al evaluar la cohesión de los componentes mediante la aplicación de los principios de cohesión, se espera identificar un nivel de cohesión alto entre los componentes.	

Elaboración Propia.

3.3. DISEÑO Y MÉTODO DE LA INVESTIGACIÓN

3.3.1. Tipo de diseño

El enfoque de investigación utilizado en esta investigación es de carácter aplicada, el cual tiene como objetivo abordar y resolver problemas concretos que se presentan en la realidad, con el propósito de encontrar una solución adecuada para una situación específica.

3.3.2. Diseño de investigación

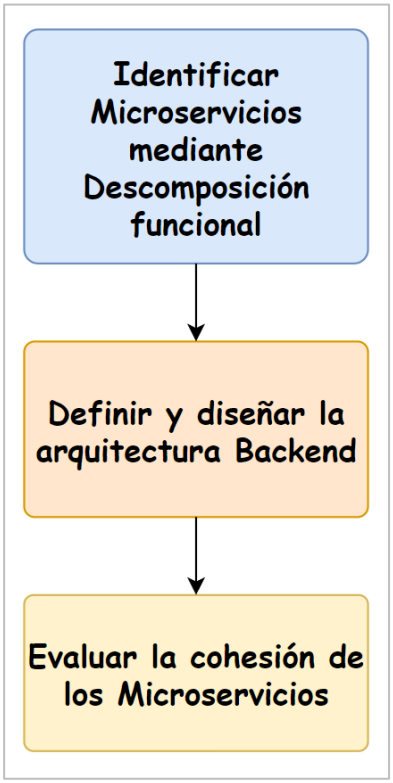
El diseño del estudio adoptado fue de tipo cuasi experimental. En este tipo de diseño, se manipula al menos una variable independiente para observar su impacto, mejora y relación con una o más variables dependientes.

3.3.3. Método

El enfoque metodológico empleado en esta investigación tuvo como objetivo analizar, implementar y evaluar la cohesión de los componentes Backend basados en Microservicios. La cohesión refiere el grado en que los componentes de un sistema están interrelacionados y colaboran de manera eficaz. Simultáneamente, se emplearon estrategias de desacoplamiento con la finalidad de mejorar la eficiencia y la calidad de dichos componentes. El propósito fundamental consistió en producir hallazgos concretos y sugerencias que puedan aplicarse en la entidad financiera con el fin de elevar la eficacia y la calidad de sus sistemas de software. Se emplearon métodos y métricas específicas para dirigir la investigación de manera sistemática y eficaz, se ilustra en la Figura 8.

Figura 8

Procedimiento metodológico.



Elaboración Propia.

Primero, se utilizó la Descomposición Funcional como el método primordial. Mediante este procedimiento, se dividió los sistemas en sus elementos fundamentales con el propósito de identificar las tareas particulares que llevan a cabo, utilizando una tabla que detalla las operaciones y sus relaciones. Este análisis se utilizó como cimiento para la segmentación y definición de los Microservicios a ser implementados. Se detalla en el apartado "*3.7.1. Identificación de microservicios utilizando descomposición funcional*".

En una segunda instancia, se puso énfasis en el diseño de la arquitectura de Microservicios. El objetivo fue identificar enfoques de diseño que sean tanto ordenados como eficientes, y que, a su vez, contribuyan a fortalecer la cohesión y a reducir el acoplamiento de estos.

Finalmente, se llevó a cabo la evaluación de la cohesión utilizando la métrica planteada por Saadati & Motameni (2014), basado en un Árbol de Subconjuntos. Esta técnica se aplicó para analizar la cohesión de los componentes de Microservicios, identificando las conexiones entre los métodos y atributos. Se logró cuantificar y representar gráficamente la cohesión en relación con la agrupación y la interrelación de los métodos. Esta métrica se fundamenta en las siguientes fórmulas:

Formula 1: Obtiene el peso del grupo **(WG)**.

$$\text{Weight of group G} = \frac{\text{Number of methods in a group}}{\text{Total number of public methods in subset tree}} \quad \textbf{(1)}$$

Formula 2: Obtiene la sumatoria de los **WG** por cada método.

$$S(M) = \sum WG(M) \quad \textbf{(2)}$$

Formula 3: Obtiene la relación de cada **S(M)** con el **MaxS(M)inTree** por cada método.

$$R(M) = \frac{S(M)}{MaxS(M)inTree} \quad \textbf{(3)}$$

Formula 4: Obtiene la cohesión de la clase.

$$Cohsion(SubsetTree) = \frac{\sum R(M)}{TM} \quad \textbf{(4)}$$

La explicación detallada del procedimiento, que implica la aplicación de estas fórmulas para evaluar la cohesión, se encuentra en la sección "*3.7.2. Medición de la cohesión y el acoplamiento*".

3.4. POBLACIÓN Y MUESTRA

3.4.1. Población

La población que constituye el foco de estudio en esta investigación es la aplicación Credirapp, la cual desempeña un papel fundamental en la ejecución del proceso de admisión de créditos. Es importante señalar que esta aplicación se presenta como una extensión al sistema monolítico ya existente.

3.4.2. Muestra

La muestra seleccionada comprende los Microservicios Backend del sistema Credirapp, abarcando todos los módulos que operan en este entorno y desempeñan una función esencial en el procesamiento de datos. La elección de esta muestra se fundamenta en la importancia y relevancia del sistema en el contexto de la entidad financiera, así como en la representatividad de los servicios elegidos para la evaluación de la cohesión y la implementación de estrategias de

desacoplamiento. A continuación, en la Tabla 3 se muestra los módulos de la aplicación Credirapp.

Tabla 3

Módulos del sistema Credirapp para el proceso de admisión de créditos.

Módulos	Descripción
Consulta de clientes	Permite la búsqueda y consulta de información relacionada con clientes. Esto incluye clientes naturales (individuos), jurídicos (empresas) y aquellos vinculados. Y adicional la información de la Superintendencia de Banca, Seguros (SBS).
Mantenimiento de clientes	Permite el mantenimiento y actualización de datos de clientes, abarcando tanto a clientes naturales como jurídicos.
Datos de la solicitud	Proporciona funcionalidades relacionadas con la solicitud de créditos. Esto incluye la presentación de solicitudes de crédito y la visualización de la posición consolidada del cliente. La posición consolidada se refiere a la situación financiera global del cliente.
Documentos virtuales	Permite la carga y gestión de documentos electrónicos asociados a las solicitudes de créditos. Estos documentos son relevantes para evaluar la solicitud.

Elaboración Propia.

3.5. MATERIALES Y EQUIPOS UTILIZADOS

En este proyecto, se emplearon una variedad de materiales y equipos técnicos con el propósito de llevar a cabo la evaluación de la cohesión entre componentes Backend basados en Microservicios. A continuación, se detallan los materiales y equipos utilizados:

Materiales:

- **Documentación Técnica:** Se emplearon documentos técnicos proporcionados por los usuarios expertos. Estos documentos comprenden detalles sobre los requisitos funcionales del sistema.
- **Herramientas de Desarrollo de Software:** Se utilizaron herramientas de desarrollo de software, incluyendo entornos de desarrollo (IDE), compiladores, depuradores para la implementación y evaluación de las estrategias de desacoplamiento.
- **Acceso a Bases de Datos:** Se contó con acceso a las bases de datos de la entidad financiera para la extracción y análisis de datos relacionados con los componentes de Microservicios y su funcionamiento.

Equipos:

- **Equipos de Computación:** Se dispuso de computadoras de trabajo de alto rendimiento con capacidades adecuadas para el desarrollo, análisis y evaluación de software. Dichos equipos fueron utilizados para la implementación de las estrategias de desacoplamiento y la implementación de los Microservicios.

La combinación de estos materiales y equipos fue esencial para llevar a cabo la evaluación de la cohesión y la aplicación de estrategias de desacoplamiento en los componentes de Microservicios. Estos recursos proporcionaron las herramientas necesarias para analizar, implementar y evaluar de manera efectiva.

3.6. MÉTODO PARA LA RECOPILACIÓN DE LOS DATOS

La recopilación de datos se centró en la metodología Scrum, aprovechando la documentación generada durante los "*sprints*" para obtener una comprensión exhaustiva de las necesidades empresariales. Este enfoque se desglosó en diversas etapas, destacando la revisión detallada de la documentación Scrum, que incluyó elementos como el backlog del producto, las historias de usuario y las reuniones de revisión y retrospectiva, junto con cualquier documentación adicional generada durante el desarrollo. Se enfatizó la traducción de esta información en requisitos funcionales para la implementación de las API, minimizando así el impacto de Scrum en el proceso de obtención de datos.

Es importante destacar que, en cumplimiento con las políticas de privacidad de la empresa, los datos empleados en esta investigación se presentaron de forma parcialmente generalizada y superficial. Se utilizaron nombres similares a la implementación real con el objetivo primordial de preservar la confidencialidad de la información sensible.

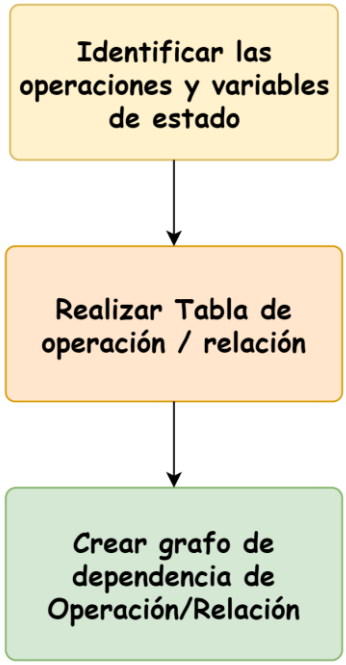
3.7. MÉTODOS PARA EL ANÁLISIS DE LOS DATOS

3.7.1. Identificación de microservicios utilizando descomposición funcional

Uno de los desafíos más grandes al crear Microservicios, está vinculado a la correcta fragmentación del sistema. Por lo general, esta tarea se realiza de manera intuitiva, confiando en la experiencia de los diseñadores. El método sistemático propuesto por Tyszberowicz et al (2018) propone la identificación de Microservicios durante las fases iniciales del proceso de diseño. Este método se fundamenta en la especificación de los requerimientos funcionales del sistema y en la descomposición funcional de estos elementos.

Figura 9

Pasos del método de descomposición funcional.



Elaboración Propia.

Se identifica las relaciones entre las operaciones que se requieren del sistema y las variables de estado que estas operaciones escriben o leen. El objetivo es crear una representación visual que clarifique las interrelaciones entre las operaciones y sus respectivas variables de estado. Las operaciones, también denominadas procesos funcionales, se vinculan con las variables de estado, conocidas como conjuntos de datos, que abarcan desde tablas en bases de datos y consultas hasta procedimientos, atributos y otros conjuntos unificados de información. Este enfoque gráfico simplifica la identificación de áreas con conexiones sólidas, señalándolas como posibles candidatas para la creación de Microservicios. La premisa fundamental es que las operaciones de cada Microservicio tengan acceso preferencial a las variables de estado específicas vinculadas a ese Microservicio (Tyszberowicz et al., 2018).

Para aplicar este método, es necesario desarrollar un modelo del sistema, que se compone de un conjunto finito de operaciones del sistema y el espacio de estados correspondiente. Las operaciones del sistema hacen referencia a las operaciones públicas, es decir, los métodos accesibles del sistema. Por otro lado, el espacio de estado engloba el conjunto de variables del sistema que almacenan información leída o escrita por las operaciones del sistema (Tyszberowicz et al., 2018).

Todas las acciones indicadas en los casos de uso se consideran operaciones, mientras que los sustantivos presentes en las descripciones de los casos de uso ofrecen una aproximación a las variables de estado del sistema. Toda esta información se documenta en una tabla de operación/relación, donde se especifica qué variables son leídas o escritas por cada operación.

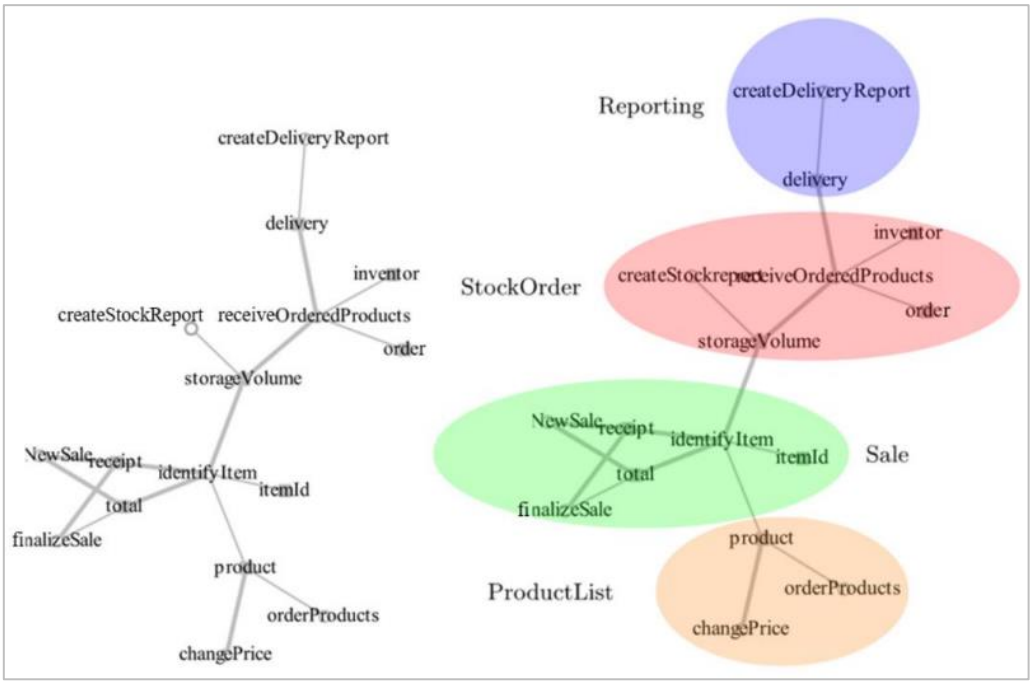
La información de la tabla de operación/relación puede transformarse en un grafo. Esta transformación posibilita la identificación de conjuntos densamente relacionados que presentan conexiones más débiles con otros conjuntos densamente relacionados. Cada uno de estos conjuntos se considera una opción favorable para ser convertido en un Microservicio, ya que comparte una cantidad limitada de información con el resto del sistema "*acoplamiento bajo*" y presenta relaciones internas densas "*cohesión alta*" (Tyszberowicz et al., 2018).

Se crea un grafo bipartito no dirigido, donde los vértices representan tanto las operaciones como las variables de estado del sistema. Las aristas establecen conexiones entre una operación y una variable de estado si y solo si la operación lee o escribe la variable. Cada arista se asocia con un peso, proponiéndose en el artículo un peso de 1 para las operaciones de lectura y un peso de 2 para las de

escritura. Esta asignación tiene como objetivo agrupar, en la medida de lo posible, aquellas operaciones que modifican una variable de estado. Se prefiere una interfaz de lectura sobre una de escritura entre los grupos relacionados. La Figura 10 Presenta los resultados obtenidos al llevar a cabo estas acciones en el caso de estudio especificado en el artículo.

Figura 10

Ejemplo de Grafo de dependencia entre Operaciones / Relaciones.



Fuente: (Tyszberowicz et al., 2018)

3.7.2. Medición de la cohesión y el acoplamiento

La cohesión y el acoplamiento son considerados como dos de las métricas más cruciales para evaluar la solidez estructural de un sistema basado en el paradigma de Programación Orientada a Objetos. Dentro del desarrollo de software orientado a objetos, la cohesión se refiere al grado en que los métodos públicos de una clase realizan tareas similares, mientras que el acoplamiento indica el nivel de dependencia de una clase con respecto a otras clases en el

sistema. Al examinar las métricas existentes relacionadas con cohesión, acoplamiento, tamaño y reutilización, se observa que todas estas métricas se basan en datos de entrada compartidos.

Con el propósito de medir la cohesión de una clase, se propone una nueva métrica que evalúa la relación entre los métodos públicos según su contribución relativa a la funcionalidad general pública de una clase. Para determinar esta contribución relativa de los métodos públicos, se introduce un nuevo concepto denominado *"árbol de subconjuntos"*. La métrica propuesta para la cohesión de clase asigna valores en un intervalo de 0 a 1 (Saadati & Motameni, 2014).

3.7.2.1. Métrica de cohesión de clases

La métrica propuesta evalúa la cohesión de una clase. Utilizando el concepto de *"árbol de subconjuntos"* para calcular esta contribución, considerando el uso compartido de atributos entre los métodos públicos de la clase. Este enfoque proporciona una medida más precisa de la cohesión, permitiendo una evaluación más detallada de la interrelación entre los métodos públicos de la clase en función de su impacto en la funcionalidad global (Saadati & Motameni, 2014).

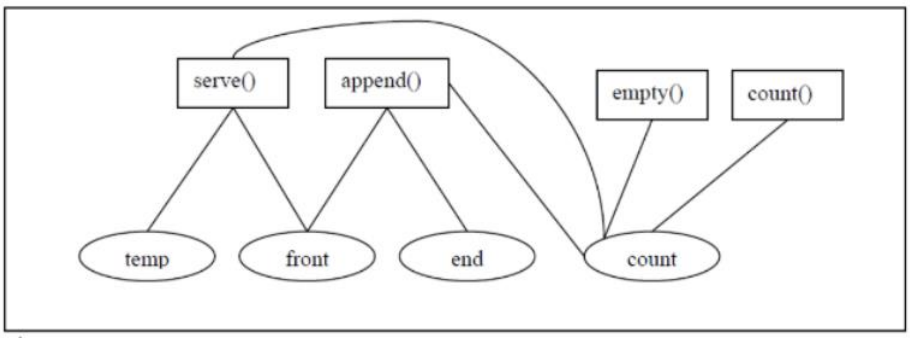
Los métodos públicos y todos los atributos de la clase ***X*** pueden ser expresados como un conjunto ***S(X)***, donde ***S(X)*** representa la unión de los conjuntos ***M(X)*** y ***A(X)***, de la siguiente manera.

$$S(X) = M(X) \cup A(X)$$

El conjunto ***M(X)*** contiene métodos públicos de la clase ***X***, mientras que el conjunto ***A(X)*** contiene todos los atributos de la clase ***X*** independientemente de su alcance y tipo.

Figura 11

Clase compuesta por métodos y atributos.



Fuente: (Saadati & Motameni, 2014)

En la Figura 11, se presenta los métodos públicos utilizando rectángulos y los atributos mediante óvalos. Una conexión entre un rectángulo y un óvalo indica el uso de un atributo por parte de un método. Por ejemplo, hay cuatro líneas que enlazan el atributo ***"count"*** con cuatro métodos distintos: ***count(), empty(), append() y serve()***. Esto se puede expresar de la siguiente manera:

G1 = {serve ()}

G2 = {serve (), append ()}

G3 = {append ()}

G4 = {serve (), append (), empty (), count ()}

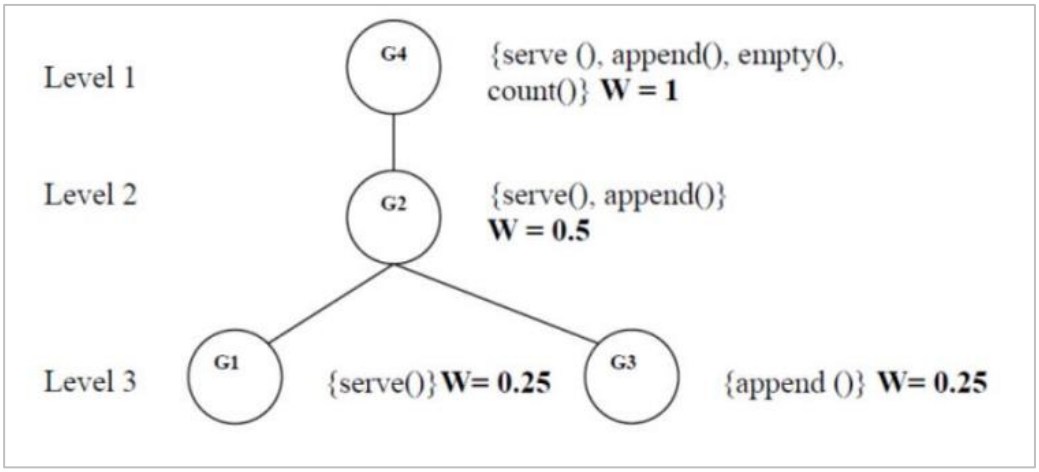
Formación de árboles de subconjuntos:

Como se ilustra anteriormente, los grupos constituyen subconjuntos, y también podemos representar dicha formación mediante un árbol de subconjuntos.

En este árbol, cada nodo representa un grupo, y su hijo representa el subconjunto de ese nodo. La Figura 12 exhibe un árbol de subconjuntos para los grupos identificados.

Figura 12

Árbol de subconjuntos representativo de los grupos de la clase.



Fuente: (Saadati & Motameni, 2014)

El árbol de subconjuntos representado en la Figura 12 ilustra la disposición de diferentes grupos en distintos niveles del árbol. El Grupo ***G4***, situado en el nivel raíz, está fuertemente vinculado con la clase, ya que incluye todos los métodos públicos. En el Nivel 2, el Grupo ***G2*** muestra una conexión comparativamente menor con la clase, mientras que los Grupos ***G1*** y ***G3*** tienen una asociación aún menor en comparación con ***G2*** y ***G4***. En la Figura 12, hemos asignado pesos ***Wi*** a todos los grupos ***Gi*** (nodos) en el árbol de subconjuntos mediante la aplicación de la **Ecuación (1)**.

$$\text{Weight of group G} = \frac{\text{Number of methods in a group}}{\text{Total number of public methods in subset tree}} \quad \textbf{(1)}$$

La anterior explicación para medir la relación de un método ***R(M)*** es ignorar el efecto de la asociación relativa de los métodos con la clase. Primero, se

tiene ***S(M)*** como la suma de pesos para un método ***M*** basado en la apariencia de ***M*** en grupos de un árbol de subconjunto expresado en la **Ecuación (2).**

$$S(M) = \sum WG(M) \quad \textbf{(2)}$$

WG(M) indica el peso ***Wi*** del grupo ***Gi*** para un método ***M*** en su presencia en el grupo ***Gi***. Si recordamos el ejemplo de la clase y consideramos los pesos del árbol de subconjuntos, los valores de ***S(M)*** para el método "***append***" se determinarán de la siguiente manera:

S (append) = WG2 (append) + WG4 (append) + WG3 (append)

S (append) = 0.50+ 1.0 + 0.25 = 1.75

La inclusión del método se produce en tres grupos, específicamente ***G2***, ***G3*** y ***G4***. Por ende, se suma los pesos de dichos conjuntos para el método ***"append"*** con el propósito de calcular ***S(M)***. Estos mismos procedimientos se aplican para calcular los valores de ***S(M)*** en los otros tres métodos de la clase.

S (count) = WG4 (count) = 1.0

S (empty) = WG4 (empty) = 1.0

S (serve) = WG1 (serve) + WG3 (serve) + WG4 (serve) = 0.50 + 0.25 + 1.0 = 1.75

Cada árbol de subconjuntos de una clase aporta a la funcionalidad general de la clase, y cada método en el árbol de subconjuntos contribuye a la funcionalidad general presentada por ese árbol, aunque esta contribución puede no ser uniforme en ambos niveles. En términos de la contribución de un método a un árbol de subconjunto, aquel con el valor ***S(M)*** más alto está más estrechamente asociado con la funcionalidad global del conjunto. En consecuencia, la relación

del método se define como una medida relativa al máximo valor de ***S(M)*** que ese método alcanza en el árbol de subconjuntos. En consecuencia, se obtiene el valor ***S(M)*** de un método ***M*** al dividirlo entre el máximo ***S(M)*** de otro método en un árbol de subconjunto específico. La fórmula para calcular la relación de los métodos públicos se muestra en la **Ecuación (3)**.

$$R(M) = \frac{S(M)}{MaxS(M)inTree} \quad \textbf{(3)}$$

Utilizando la fórmula, se obtienen los siguientes valores relativos para todos los métodos de la clase. El valor máximo de ***S(M)*** para todos los métodos en el árbol de subconjuntos es 1.75.

R (append) = 1.75/1.75 = 1.0

R (count) = 1.0/1.75 = 0.57

R (empty) = 1.0/1.75 = 0.57

R (serve) = 1.75/1.75 = 1

Para evaluar la cohesión del árbol de subconjuntos, se calcula el promedio de la medida de la relación ***R(M)*** de todos los métodos en el árbol de subconjuntos utilizando la siguiente **Ecuación (4)**.

$$Cohsion(SubsetTree) = \frac{\sum R(M)}{TM} \quad \textbf{(4)}$$

TM representa el total de métodos en el árbol de subconjuntos. Para la Clase, obtendremos los resultados siguientes:

Cohesion (Class Queue) = (1.0+0.57+0.57+1.0) / 4 = 0.785

En el caso de la clase Cola, se tiene únicamente de un árbol de subconjunto; por lo tanto, la cohesión del árbol de subconjunto equivale a la cohesión de la clase. La cohesión de la clase se caracteriza por este valor, siendo considerada como una clase cohesionada. En el caso de tener varias clases, la cohesión se calcula a través de la media aritmética. Esta métrica demuestra ser más eficaz que aquellas que solo evalúan la cohesión de una clase considerando la relación entre sus métodos públicos.

CAPÍTULO IV

RESULTADOS Y DISCUSIÓN

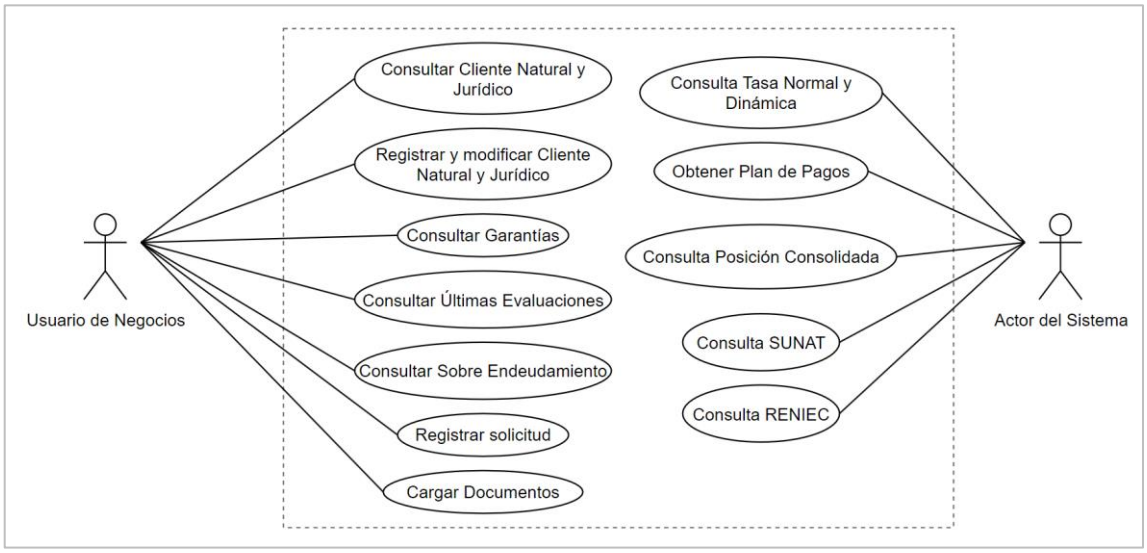
4.1. DESCRIPCIÓN DEL CASO DE ESTUDIO

Este estudio se enfocó en analizar, implementar y evaluar la cohesión de los componentes Backend basados en Microservicios para la aplicación Credirapp de la entidad financiera Los Andes, esta aplicación se enfoca específicamente en el proceso de admisión de créditos, buscando mejorar, optimizar y agilizar la gestión de solicitudes dentro de la entidad financiera, beneficiando tanto a los clientes como a la institución.

La situación actual en la institución se caracteriza por operar en un entorno monolítico con limitadas capacidades de escalabilidad en diversas plataformas. Debido a esta restricción, la aplicación debe cumplir con estándares de calidad elevados, lo que impulsa la evaluación de la cohesión y la priorización de un bajo acoplamiento en el enfoque de desarrollo. Para abordar este desafío, se optó por implementar arquitecturas de Microservicios, guiadas por los principios SOLID y estructuradas mediante una arquitectura limpia de capas “Onion Architecture”. Por parte del área usuaria, se adoptó el marco ágil Scrum. Al integrar desarrolladores y expertos financieros, colaboraron en “*sprints*” cortos. La documentación generada durante esos periodos proporcionó una comprensión completa de las necesidades y el alcance de la aplicación. A continuación, la Figura 13 presenta los casos de uso identificados en la documentación, junto con los actores del sistema correspondientes.

Figura 13

Diagrama de casos de uso identificados para el sistema Credirapp.



Elaboración Propia.

Interpretación: Se presenta un caso de uso que abstrae las funcionalidades del sistema, centrándose especialmente en el proceso de admisión crediticia. Por un lado, se tiene el Actor "*Usuario de Negocios*", el cual realiza ciertas acciones identificadas para dicho propósito. Por otra parte, el "*Actor del Sistema*" complementa procesos automáticos internos, con el fin de mejorar la experiencia del usuario.

4.2. OBJETIVO ESPECÍFICO 01, IDENTIFICACIÓN DE MICROSERVICIOS USANDO DESCOMPOSICIÓN FUNCIONAL

Se aplicó el método de descomposición funcional propuesto por Tyszberowicz et al. (2018). Esta identifica las operaciones del sistema y las variables de estado mediante la utilización de casos de uso. Todas las acciones indicadas en los casos de uso se consideran operaciones o procesos funcionales. Es importante resaltar que de un caso de uso se pueden obtener varios procesos funcionales y un proceso funcional puede interactuar con varios grupos de datos.

En primer lugar, las operaciones del sistema fueron obtenidas a través de los casos de uso descritos en la Figura 13. Estas operaciones se analizaron y desglosaron, interpretándolas como procesos funcionales relevantes. Posteriormente, para definir las variables de estado o también conocidos como grupos de datos, se identificaron a través de la lectura y escritura en la base de datos, considerando los procesos funcionales y la presencia de sustantivos en las descripciones de los casos de uso. Es importante destacar que, según este método, se asigna un peso de 1 a las operaciones de lectura y un peso de 2 a las operaciones de escritura. Esta información fue registrada y analizada a través de una tabla de operación/relación, que identificaron estas interacciones.

A continuación, se logró identificar los siguientes procesos funcionales y grupos de datos, Tabla 4 y Tabla 5, teniendo los PFs y GDs se identificaron los procesos de lectura y escritura mediante la tabla de operación / relación presentada en la Tabla 6.

Tabla 4

Tabla de procesos funcionales.

	Procesos funcionales
PF1	Consultar Cliente
PF2	Consultar Vinculados
PF3	Registrar Cliente Natural
PF4	Actualizar Cliente Natural
PF5	Consultar Cliente Jurídico
PF6	Registrar Cliente Jurídico
PF7	Actualizar Cliente Jurídico
PF8	Buscar Clientes
PF9	Consultar tasa dinámica campaña
PF10	Consultar Tasa
PF11	Consultar Garantías de cliente y cónyuge
PF12	Consultar Sobre Endeudamiento
PF13	Simulador plan de Pagos
PF14	Consultar ultimas evaluaciones
PF15	Registrar Solicitud de crédito
PF16	Consultar Posición Consolidada
PF17	Cargar documentos
PF18	Actualizar catálogos
PF19	Consultar SUNAT
PF20	Consultar RENIEC

Elaboración Propia.

Tabla 5

Tabla de grupos de datos.

	Grupos de datos
GD1	Identificador de Cliente
GD2	Información de cliente Natural
GD3	Información de cliente Jurídico
GD4	Vinculado
GD5	Información SBS
GD6	Información Deudas RCC
GD7	Productos
GD8	Créditos
GD9	Garantías Crediticias
GD10	Evaluación Crediticia
GD11	Ofertas
GD12	Solicitud de crédito
GD13	Estado Solicitud
GD14	Configuración de Tasas
GD15	Configuración de Tasas Dinámicas
GD16	Plan De pagos
GD17	Archivo
GD18	Tablas Tipo
GD19	SUNAT
GD20	RENIEC
GD21	Saldos RCC
GD22	Cliente RCC
GD23	Plan Contable
GD24	Parámetros de Sobre Endeudamiento

Elaboración Propia.

Tabla 6

Matriz de Operación / Relación del sistema.

PF / GD	GD1	GD2	GD3	GD4	GD5	GD6	GD7	GD8	GD9	GD10	GD11	GD12	GD13	GD14	GD15	GD16	GD17	GD18	GD19	GD20	GD21	GD22	GD23	GD24
PF1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF2	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF3	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF4	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF5	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF6	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF7	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF8	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF9	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
PF10	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
PF11	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PF12	1	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1
PF13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
PF14	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
PF15	1	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0
PF16	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
PF17	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2	0	0	0	0	0	0	0
PF18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
PF19	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
PF20	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

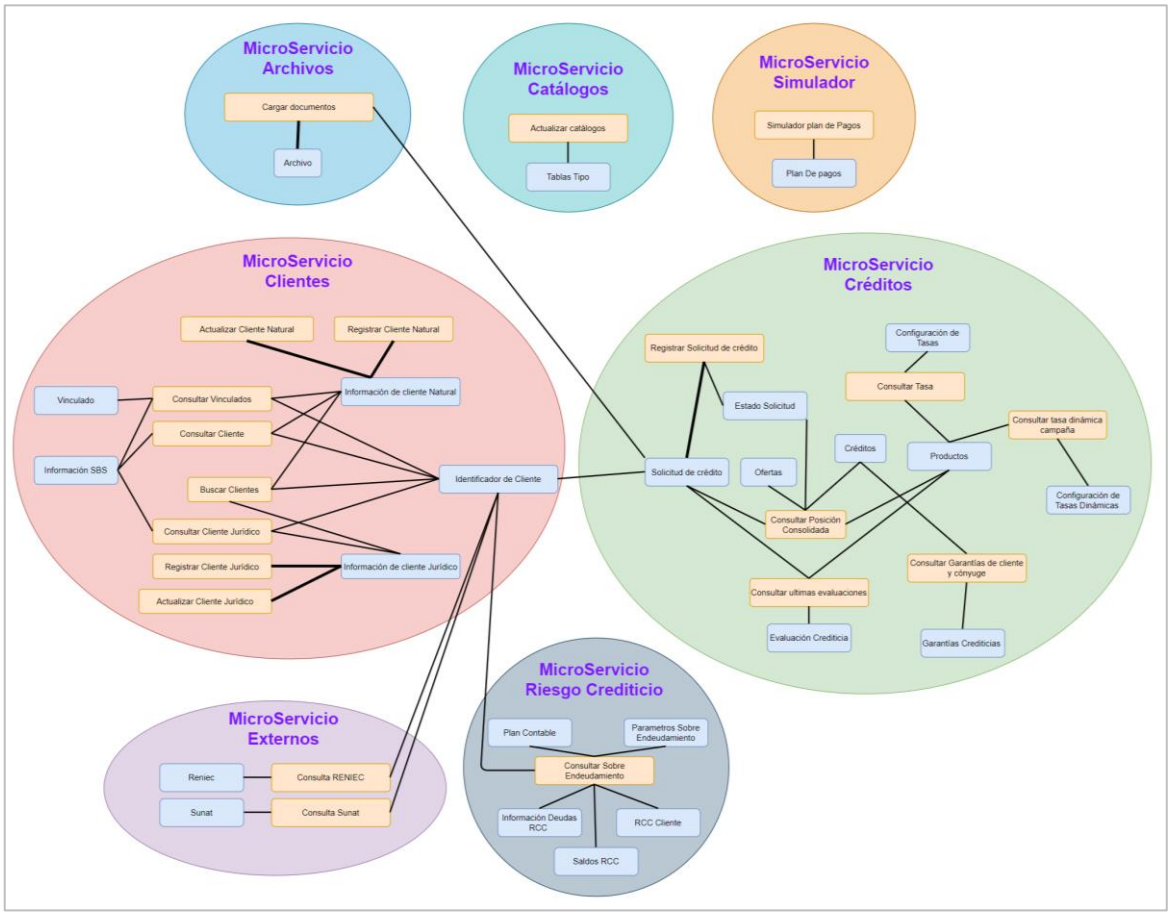
Elaboración Propia.

Los resultados obtenidos en la Tabla 6, reflejan de manera clara las interacciones entre los diversos procesos funcionales y los respectivos grupos de datos. Cada celda de la tabla indica la presencia o ausencia de relación entre una operación y un grupo de datos, proporcionando una visión detallada de la estructura funcional del sistema. Por ejemplo, para ***PF1*** existe una relación de lectura con ***GD1***, ***GD2***, ***GD5*** y para ***PF3*** y ***PF4*** una relación de escritura con ***GD2***. Este enfoque matricial permite identificar patrones y conexiones significativas dentro del sistema.

Posteriormente, se creó un grafo para visualizar las interrelaciones entre operaciones del sistema y variables de estado. Esta representación gráfica facilita la identificación de conjuntos densamente relacionados que presentan conexiones más tenues con otros conjuntos igualmente densos. Cada uno de estos conjuntos se considera propicio para convertirse en un Microservicio, dado que comparte una cantidad limitada de información con el resto del sistema, mostrando un "*acoplamiento bajo*", y presenta relaciones internas densas, es decir, una "*cohesión alta*". Dicho grafo se exhibe en la Figura 14.

Figura 14

Gráfico de dependencia de operación/relación del sistema.

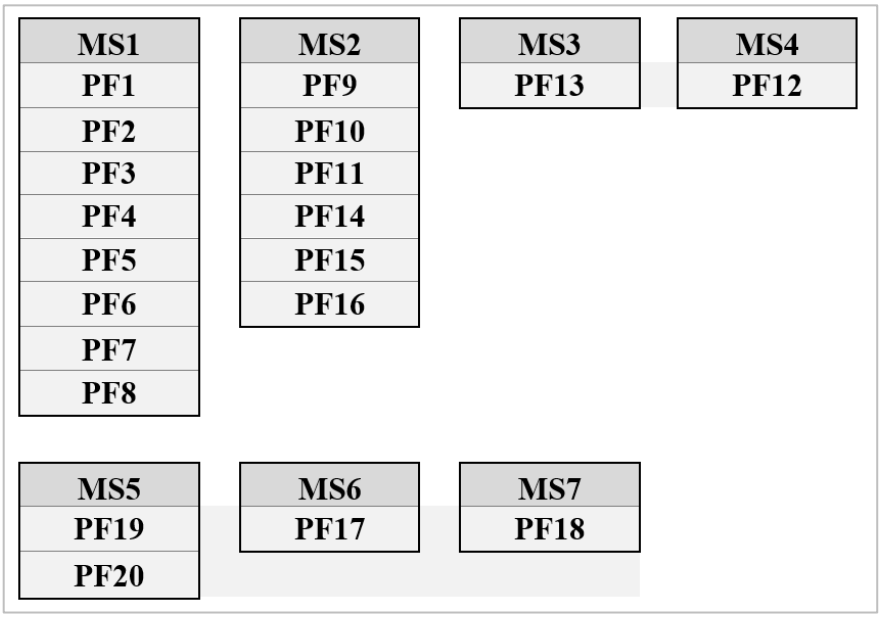


Elaboración Propia.

Esta representación gráfica permitió identificar conjuntos densamente relacionados, destacando como opciones propicias para conformar Microservicios con bajo acoplamiento y relaciones internas sólidas. La Figura 15 presenta la agrupación de procesos funcionales densamente relacionados.

Figura 15

Grupos Identificados como Microservicios.



Elaboración Propia.

La aplicación rigurosa de la descomposición funcional, siguiendo la metodología de Tyszberowicz et al. (2018) condujo a la precisa identificación de siete Microservicios clave, detallados en la Tabla 7. Este enfoque detallado proporcionó una clara segmentación de los procesos funcionales y grupos de datos, facilitando la identificación de sistemas más modulares y adaptables. Estos resultados subrayan la importancia estratégica de la descomposición funcional en el diseño de una arquitectura de Microservicios.

Tabla 7

Microservicios identificados para el sistema.

MICROSERVICIOS	
MS1	Clientes
MS2	Créditos
MS3	Simulador
MS4	Riesgo Crediticio
MS5	Externos
MS6	Archivos
MS7	Catálogos

Elaboración Propia.

DISCUSIÓN

La descomposición funcional siguiendo la metodología de Tyszberowicz et al. (2018) posibilitó la identificación precisa de siete Microservicios clave al visualizar las interrelaciones entre procesos funcionales y grupos de datos. Este enfoque detallado resalta la importancia estratégica de la descomposición funcional, al permitir una segmentación clara que da lugar a sistemas más modulares y adaptables. En contraste, la tesis de Pedraza Coello (2021) titulada "*método DISC: separando sistemas en Microservicios*" presentó un enfoque organizado en cinco etapas que abordó perspectivas de dominio, infraestructura, seguridad, calidad y la resolución de contradicciones. Este método se respaldó en criterios de acoplamiento y estándares COSMIC. Dichos estándares proveen reglas y directrices para medir el tamaño funcional del software, considerando funciones específicas que facilitaron la evaluación de la granularidad y

acoplamiento. En esa investigación se logró identificar cinco Microservicios a través del método DISC.

La comparación entre ambas investigaciones revela la diversidad de enfoques: mientras que la descomposición funcional se enfoca en las relaciones entre operaciones y variables de estado, el método DISC va más allá al incorporar análisis detallados de dominio, infraestructura, seguridad y calidad. Ambas metodologías subrayan la importancia de considerar los requisitos funcionales en la segmentación de Microservicios. Este contraste ofrece perspectivas complementarias, enriqueciendo las posibilidades para futuras investigaciones y proporcionando aplicaciones prácticas valiosas en el diseño efectivo de arquitecturas de Microservicios.

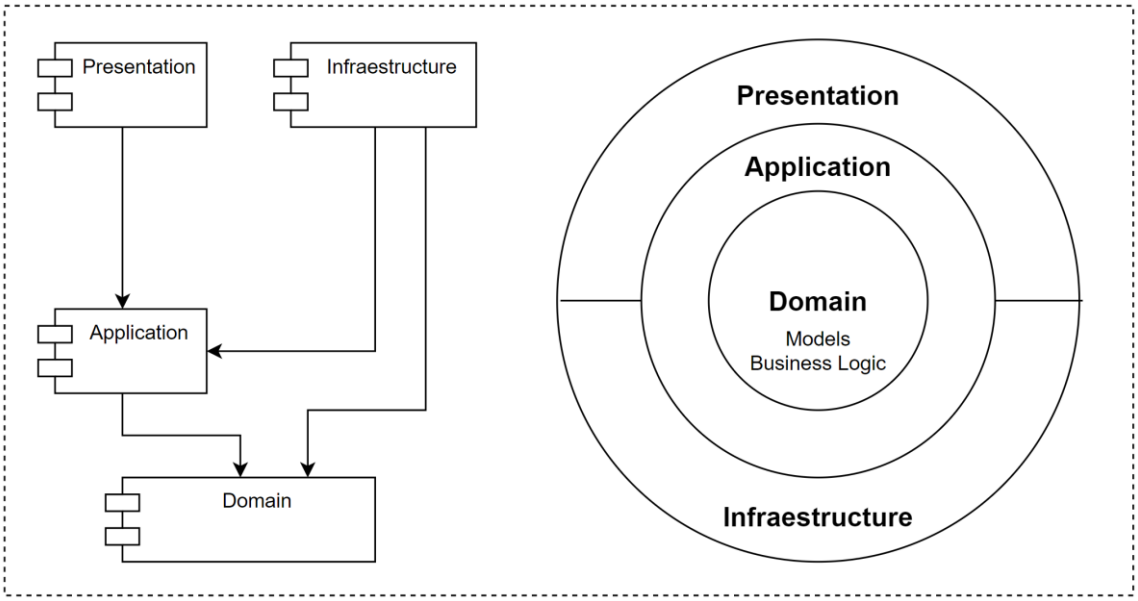
4.3. OBJETIVO ESPECÍFICO 02, DISEÑO DE LA ARQUITECTURA DE MICROSERVICIOS

Se implementó la arquitectura Onion Architecture en cada Microservicio identificado, ya que ofrece una estructura sólida que simplifica el desarrollo, mejora la escalabilidad y facilita la mantenibilidad de los Microservicios. Para el diseño se eligió considerar la implementación propuesta por Taylor (2023) ilustrada mediante un diagrama de arquitectura en capas y mediante círculos concéntricos que muestra las responsabilidades y roles específicos de cada proyecto. Esta arquitectura tuvo el propósito de maximizar la cohesión y minimizar el acoplamiento, factores cruciales para el desarrollo eficiente y sostenible.

La implementación se fundamentó en consideraciones específicas de los principios SOLID. Dichos principios proporcionaron directrices claras para alcanzar un diseño de software más sólido, flexible y fácil de mantener. A continuación, en la Figura 16 se presenta el modelo arquitectónico implementado.

Figura 16

Modelo de Arquitectura para el desarrollo del sistema.



Elaboración Propia.

Este modelo se compuso por cuatro capas *"Domain, Application, Presentation, e Infrastructure"*. Dichas capas tienen la necesidad de ofrecer una estructura modular para la aplicación Credirapp, logrando simplificar la organización del código y manteniendo consistencia con los principios SOLID, a su vez fomentando un diseño de software robusto y fácilmente mantenible. A continuación, se describe el propósito de cada una de estas capas en relación con los principios SOLID:

4.3.1. Capa de dominio

En la capa de Dominio, también conocida como "*Domain*", se definieron las entidades, interfaces de clase y reglas de negocio fundamentales que respaldan la funcionalidad principal de los Microservicios. Esta estrategia garantizó que la capa de dominio opere de manera independiente, desvinculándose de las capas externas y de tecnologías o Frameworks específicos, lo que, en el ámbito financiero, resulta fundamental por varias razones:

- En primer lugar, al mantener la lógica de negocio cohesiva y desacoplada, la adaptación a cambios normativos se vuelve más ágil. Dado que este sector financiero está sujeto a modificaciones frecuentes en regulaciones y políticas, contar con una capa de dominio flexible facilita la incorporación de ajustes necesarios para cumplir con los requisitos normativos.
- En segundo lugar, esta independencia tecnológica permite una evolución más fluida de los servicios financieros. La entidad financiera está sujeta a cambios tecnológicos, variaciones en la demanda del mercado y nuevas estrategias comerciales. La capa de dominio desacoplada brinda la adaptabilidad y escalabilidad sin comprometer la integridad de la lógica de negocio.
- Por último, la adopción de una arquitectura limpia en la capa de dominio facilita la rápida incorporación de nuevas funcionalidades, siendo crucial para la entidad financiera. Esta flexibilidad no solo proporciona una ventaja competitiva, sino también la capacidad para adaptarse rápidamente a las demandas del mercado y superar las expectativas de sus clientes.

La capa de Dominio aplicó los siguientes Principios SOLID:

- **Responsabilidad Única (SRP):** La capa de dominio encapsula la lógica de negocio, aplicando SRP al centrarse en la responsabilidad única de modelar y ejecutar las reglas empresariales.
- **Inversión de Dependencias (DIP):** Al definir interfaces abstractas para las dependencias externas, la capa de dominio invierte las dependencias, permitiendo la sustitución y adaptación sin afectar la lógica central.

4.3.2. Capa de aplicación

La capa de aplicación, también denominada "*Application*", desempeño un papel crucial al coordinar la interacción entre la capa de dominio y la capa de infraestructura. En esta capa residen los casos de uso, encargados de orquestar la lógica del negocio y los flujos de trabajo esenciales para la implementación efectiva de las funcionalidades del sistema. Este enfoque ofrece beneficios adicionales:

- En primer lugar, al centralizar la lógica de coordinación en la capa de aplicación, se logra una mayor cohesión y mantenibilidad. Dado que la institución financiera opera en un entorno altamente regulado y dinámico, tener una capa dedicada para la orquestación de la lógica facilita la adaptación a cambios normativos y actualizaciones de políticas, sin comprometer la integridad de la lógica de dominio.
- En segundo lugar, la capa de aplicación proporcionó una interfaz clara y definida para la interacción entre la lógica de dominio y la infraestructura. Esto resulta particularmente valioso para la entidad financiera, donde la claridad en la gestión de procesos y flujos de trabajo es esencial para garantizar la precisión y seguridad en operaciones críticas.

La capa de aplicación aplicó los siguientes principios SOLID:

- **Abierto/Cerrado (OCP):** La capa de aplicación permite la extensión de funcionalidades a través de servicios de aplicación, cumpliendo con el principio al admitir adiciones de servicios sin modificar código existente.

- **Inversión de Dependencias (DIP):** Al depender de abstracciones en lugar de implementaciones concretas, la capa de aplicación aplica la inversión de dependencias, facilitando la adaptación a cambios.

4.3.3. Capa de infraestructura

La Capa de Infraestructura, también denominada como "*Infrastructure*", se encargó de las implementaciones relacionadas con las bases de datos internas y externas del sistema, teniendo como responsabilidad única la obtención de datos. Para esta integración en cada Microservicio, se priorizó una fuente de datos única para garantizar la consistencia, eficiencia y seguridad en las transacciones de tablas. Esta capa puede conceptualizarse como un conjunto de piezas de lego interconectadas para proporcionar las implementaciones concretas necesarias para el funcionamiento del sistema. Cada pieza, como en un conjunto de lego, tiene una función específica, como acceso a bases de datos, interacción con servicios externos. Estas implementaciones modulares se ensamblan de manera cohesiva para construir la base sobre la cual se apoyan las capas superiores, permitiendo una arquitectura flexible y desacoplada. Esta analogía refleja la idea de que las capas de infraestructura actúan como componentes intercambiables que pueden ser combinados y adaptados según las necesidades. Esta capa adquiere especial relevancia por diversas razones:

- En primer lugar, la gestión eficiente de bases de datos es crítica en el sector financiero, donde la integridad y disponibilidad de la información son prioritarias. La Capa de Infraestructura, al asumir la responsabilidad de estas implementaciones específicas, garantiza la granularidad y alta cohesión en la obtención de datos debidamente segmentados. Esta flexibilidad facilitó la

adaptación y optimización del sistema de almacenamiento de datos, garantizando cumplimiento con requisitos y estándares para un rendimiento óptimo.

- En segundo lugar, en esta integración se consideró la importancia crítica de las transacciones de base de datos en la entidad financiera, constantemente se ejecutan diversas transacciones en tablas específicas. La infraestructura en todos los Microservicios implementados consume una única fuente de datos. Este enfoque unificado optimiza la coherencia y la integridad de la información de la entidad, asegurando que todos los servicios compartan y actualicen datos de manera consistente. Además, permite una gestión más eficiente de las transacciones, esenciales donde la precisión y la consistencia de la información son cruciales.
- Por último, la reducción del acoplamiento en la gestión de dependencias es esencial para la seguridad y estabilidad de los servicios financieros. Al diseñar conjuntos de datos con bajo acoplamiento y alta cohesión, se minimiza el impacto de posibles cambios o actualizaciones en las dependencias, garantizando la continuidad operativa y la capacidad de respuesta.

La capa de Infraestructura aplicó los siguientes Principios SOLID:

- **Inversión de Dependencias (DIP):** Esta capa invierte las dependencias al basarse en abstracciones en lugar de implementaciones concretas, lo que posibilita realizar cambios en la infraestructura sin afectar a las capas superiores.
- **Responsabilidad Única (SRP):** Se implementa al encapsular la ejecución técnica y la interacción externa, preservando la capa principal (Dominio) en su función específica.

4.3.4. Capa de presentación

La Capa de Presentación, también conocida como "*Presentation*", se especializa en la interacción externa a través de Endpoints. Esta capa asume la responsabilidad de facilitar la presentación de datos. La separación estratégica de esta capa asegura la modularidad y la capacidad de adaptarse a diversas interfaces, aspectos cruciales para la entidad por varias razones:

- En primer lugar, la Capa de Presentación permitió una interacción externa coherente y adaptativa a diferentes interfaces de usuario. En la entidad, la capacidad de adaptarse a diversas interfaces garantiza una presentación de datos uniforme y eficiente, independientemente de la plataforma o dispositivo utilizado.
- En segundo lugar, al separar la presentación de datos en una capa, se logra una mayor mantenibilidad y escalabilidad. En este contexto, donde la oferta de servicios digitales y la evolución tecnológica son constantes, esta modularidad permite introducir cambios y mejoras en la interfaz sin afectar la lógica de negocio subyacente, facilitando así la adaptación a nuevas tendencias y necesidades del usuario.

La capa de Presentación aplicó los siguientes Principios SOLID:

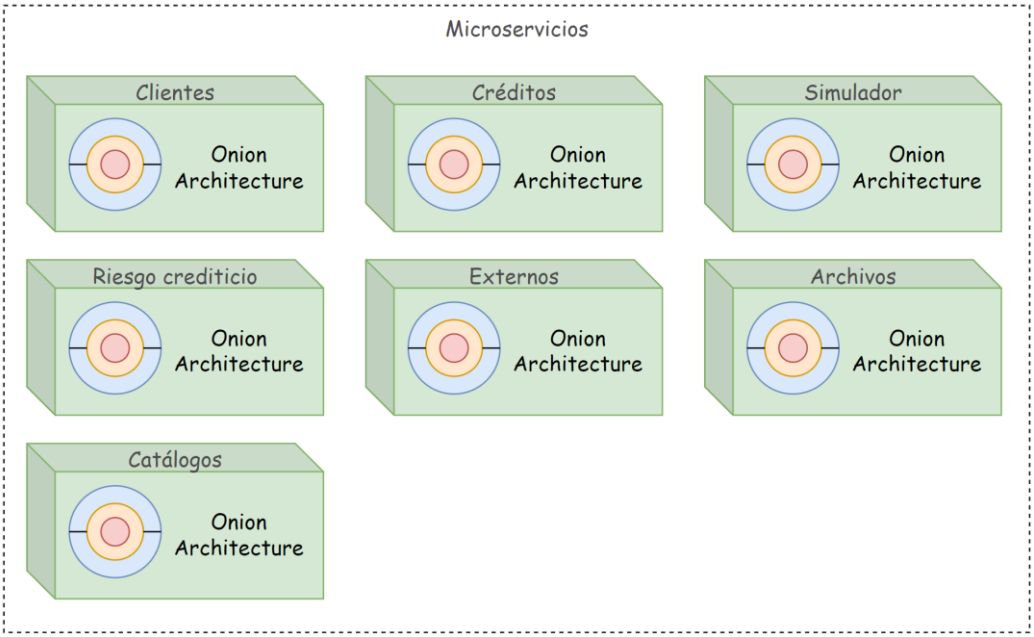
- **Segregación de Interfaces (ISP):** En esta capa, se establecen interfaces específicas según las necesidades de presentación, evitando la sobrecarga de interfaces y garantizando la adaptabilidad del sistema.
- **Inversión de Dependencias (DIP):** Al depender de abstracciones en lugar de detalles concretos de la implementación.

4.3.5. Integración del modelo arquitectónico en los microservicios

Finalmente, el modelo de arquitectura Onion Architecture se aplicó en cada uno de los siete Microservicios previamente identificados: clientes, créditos, simulador, riesgo crediticio, externos, archivos y catálogos. Se justifica por su capacidad inherente para fomentar la modularidad y escalabilidad en el diseño del sistema. Cada Microservicio, representado por varias capas en la arquitectura, encapsula sus funcionalidades específicas y datos asociados, permitiendo una clara separación de responsabilidades. En el contexto de Onion Architecture, la capa más interna contiene la lógica de negocio central, mientras que las capas externas representan aspectos como la interfaz de usuario, servicios externos y acceso a datos. Esta estructura favorece la flexibilidad y mantenibilidad al facilitar la modificación o sustitución de componentes específicos sin afectar el resto del sistema. Además, al adherirse a los principios SOLID, se maximiza la cohesión dentro de cada Microservicio y se minimiza el acoplamiento, garantizando así una implementación eficiente y sostenible. En resumen, la adopción de Onion Architecture en cada Microservicio proporciona una base robusta para la evolución y adaptabilidad de la entidad financiera Los Andes. La Figura 17 exhibe la integración de la Onion Architecture en cada uno de los Microservicios identificados.

Figura 17

Representación de Onion Architecture en cada Microservicio.



Elaboración Propia.

DISCUSIÓN

La implementación de Microservicios en el sistema Credirapp de la entidad financiera Los Andes se fundamentó en Onion Architecture y se guio por los principios SOLID. Cada capa cumplió con su propósito específico, desde la capa de Dominio que encapsula la lógica de negocio hasta la capa de Presentación que facilita la interacción externa. La independencia y desacoplamiento destaca la importancia de capas específicas para fortalecer la cohesión y garantizar la agilidad ante cambios funcionales y normativos. Comparativamente, con la investigación de Nieto Sánchez (2017), "*Modelo de Arquitectura de Software para Aplicaciones iOS basado en Clean Architecture*", se diseñó un modelo evolutivo específico para aplicaciones iOS, basándose en la filosofía Clean Architecture. Este modelo resalta por su enfoque en modularidad, reutilización, patrones de diseño y los principios SOLID, respaldado por una metodología adaptada del Test Driven Development (TDD), o Desarrollo Guiado por Pruebas en español. Esta

investigación refuerza estos principios, evidenciando beneficios similares en términos de desarrollo eficiente y mantenimiento simplificado. Ambos estudios convergen en la relevancia de principios SOLID para asegurar la calidad y sostenibilidad en distintos contextos, respaldando la idea de que cada arquitectura bien fundamentada es esencial para el éxito duradero del sistema de software.

4.4. OBJETIVO ESPECÍFICO 03, EVALUACIÓN DE LA COHESIÓN DE LOS COMPONENTES BACKEND EN MICROSERVICIOS

Se creó una aplicación web que integra la lógica necesaria para evaluar la cohesión en cada microservicio implementado, cumpliendo con la métrica propuesta por Saadati & Motameni (2014). El propósito central de este estudio es evaluar y determinar el grado de cohesión presente en los Microservicios. Posteriormente, se exhiben los resultados obtenidos de esta evaluación.

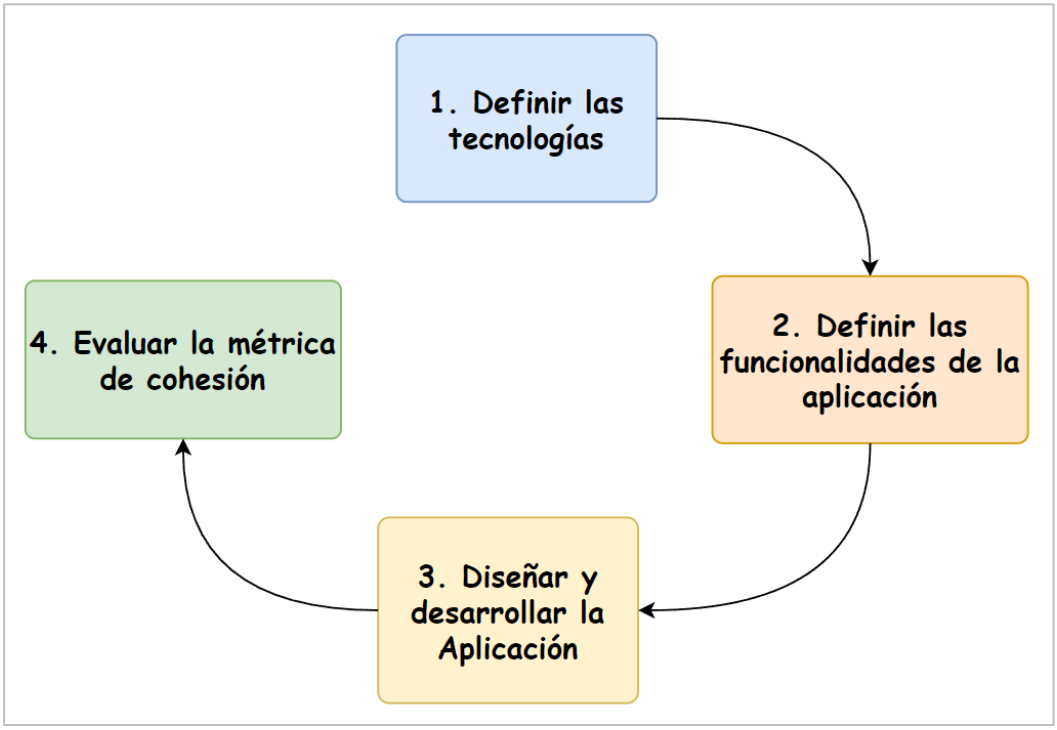
4.4.1. Desarrollo de una aplicación web para la evaluación de cohesión

El propósito principal de la aplicación denominada "*Diseñador de objetos e interrelaciones*", fue desarrollar una herramienta especializada para posibilitar la evaluación automatizada y eficiente de la cohesión de clases en aplicaciones fundamentadas en Microservicios. La aplicación incorporó la métrica de Saadati & Motameni (2014)**,** dicha métrica evalúa la relación entre los métodos públicos según su contribución relativa a la funcionalidad general pública de una clase. Para determinar esta contribución relativa de los métodos públicos, se introduce un nuevo concepto denominado *"árbol de subconjuntos"*. La métrica propuesta para la cohesión de clase asigna valores en un intervalo de 0 a 1. La aplicación busca mejorar la toma de decisiones arquitectónicas y fomentar una práctica

continua de desarrollo de software orientada a la calidad y eficiencia en entornos de arquitecturas de Microservicios.

Figura 18

Procedimiento para la implementación de la aplicación web.



Elaboración Propia.

4.4.1.1. Tecnologías utilizadas

A continuación, se describen los aspectos técnicos que componen la infraestructura y el desarrollo de la aplicación en la Tabla 8.

Tabla 8

Aspectos Técnicos de la implementación de la aplicación web.

Aspecto	Tecnología / Descripción
Lenguajes de Programación	TypeScript (Node.js)
Framework	ReactJS (TypeScript)
Base de Datos	Cloud Firestore (Firebase - No relacional)
Herramientas de Desarrollo	Visual Studio Code (Editor de código), Git (Control de versiones)
Plataforma de Implementación	Desplegado en Vercel: Esta plataforma en la nube facilita el desarrollo web al integrarse con GitHub y automatizar eficientemente el despliegue continuo.
Arquitectura	Clean Architecture (Frontend)
Escalabilidad y Rendimiento	Uso de cachés para optimizar la respuesta, optimización de consultas de base de datos.

Elaboración Propia.

La implementación de la aplicación se llevó a cabo bajo el marco de tecnologías avanzadas que se alinean estratégicamente con las necesidades y desafíos del proyecto. En primer lugar, se optó por el uso de TypeScript en el entorno Node.js para el desarrollo del Backend, proporcionando beneficios significativos en términos de tipado estático y escalabilidad del código. En cuanto al Frontend, la elección del Framework ReactJS, también implementado en TypeScript, garantiza una estructura robusta y modular para la interfaz de usuario, facilitando así su desarrollo y mantenimiento.

En el ámbito de la persistencia de datos, se adoptó Cloud Firestore de Firebase como la base de datos no relacional, aprovechando su flexibilidad y

capacidad de escalabilidad para gestionar eficientemente los datos de la aplicación. Además, se utilizaron herramientas de desarrollo líderes como Visual Studio Code para la edición de código y Git para el control de versiones, asegurando un flujo de trabajo eficiente y colaborativo.

Para la implementación y despliegue continuo, la aplicación se desplegó en Vercel, una plataforma en la nube que simplifica el proceso al integrarse con GitHub, facilitando así la automatización del despliegue. Esta elección estratégica se alinea con la eficiencia y agilidad requeridas en el desarrollo web contemporáneo.

En cuanto a la arquitectura, se adoptó Clean Architecture en el Frontend, buscando una clara separación de responsabilidades entre las capas de la aplicación. Esta decisión no solo facilita la comprensión del código, sino que también favorece la adaptabilidad y mantenimiento a largo plazo.

En términos de escalabilidad y rendimiento, se implementaron estrategias de caché para optimizar la respuesta y se realizaron optimizaciones específicas en las consultas de la base de datos. Estas medidas aseguran una experiencia de usuario eficiente y una gestión eficaz de los recursos, contribuyendo a la viabilidad y rendimiento sostenible del sistema. En resumen, la selección cuidadosa de tecnologías y enfoques evidencia una atención meticulosa a la calidad y eficacia en el desarrollo de la aplicación, respaldando así los objetivos y requerimientos establecidos para este proyecto en la evaluación de la cohesión entre objetos e interrelaciones en Microservicios.

4.4.1.2. Funcionalidades implementadas

Las funcionalidades se centran en simplificar la evaluación de la cohesión en Microservicios. A continuación, se han implementado las siguientes características:

- **Creación y gestión de proyectos:** Esta funcionalidad posibilitó a los usuarios la creación y gestión de proyectos en la plataforma. Los usuarios tenían la capacidad de iniciar nuevos proyectos proporcionando información esencial, como el nombre del proyecto y su descripción. Además, se les brindaba la capacidad de acceder a proyectos anteriores, modificar detalles existentes o eliminarlos, permitiendo así una gestión completa y flexible de los proyectos en la plataforma.
- **Creación de nodos:** Esta característica permitió la creación y gestión de bloques visuales denominados "*nodos*". Cada nodo representó un elemento programático, abarcando capas lógicas, Microservicios, Endpoints, clases, métodos, atributos, consultas a bases de datos, procedimientos de bases de datos, funciones de bases de datos y tablas de bases de datos. La creación de un nodo implicó la especificación de información crucial, como el nombre, descripción y tipo de nodo. Asimismo, se ofreció la capacidad de eliminar nodos, proporcionando así una gestión completa y flexible de la representación visual de estos elementos en el sistema.
- **Administración de la relación entre nodos:** Esta funcionalidad esencial capacitó la interconexión de nodos mediante relaciones de dependencia y parentesco. En el ámbito de la programación, la relación de dependencia se materializó en la conexión entre métodos y atributos, donde estos últimos implementaban funciones y dependían de fuentes de datos específicas. Este

vínculo también se extendió a través del parentesco, representando una relación jerárquica en la cual un nodo de tipo clase podía tener nodos de tipo método, estableciendo así una relación estructurada donde los métodos eran considerados "*hijos*" dentro de la jerarquía de objetos. La implementación de esta funcionalidad no solo facilitó la modelación precisa de las dependencias entre elementos, sino que también contribuyó a la comprensión y organización eficientes del sistema, fundamentales para el diseño cohesivo y la evaluación de interrelaciones en el contexto de Microservicios.

- **Integración de la métrica de Cohesión:** La métrica de cohesión propuesta por Saadati & Motameni (2014) fue integrada como componente fundamental del sistema. Esta métrica evaluó el nivel de cohesión en las interrelaciones entre métodos y atributos dentro de una clase, proporcionando una puntuación en una escala de 0 a 1. La representación de esta métrica pudo expresarse en forma de porcentajes, donde un puntaje más alto indicaba una fuerte cohesión en la clase. La implementación de esta funcionalidad permitió una evaluación cuantitativa precisa de la cohesión, brindando a los usuarios una comprensión clara y detallada del grado de interrelación entre los componentes de la clase.
- **Representación de resultados:** Esta funcionalidad clave se encargó de generar representaciones visuales detalladas de la evaluación de cohesión. Su objetivo era proporcionar a los desarrolladores y arquitectos una herramienta visualmente enriquecedora para comprender la complejidad y las interrelaciones dentro de los Microservicios.

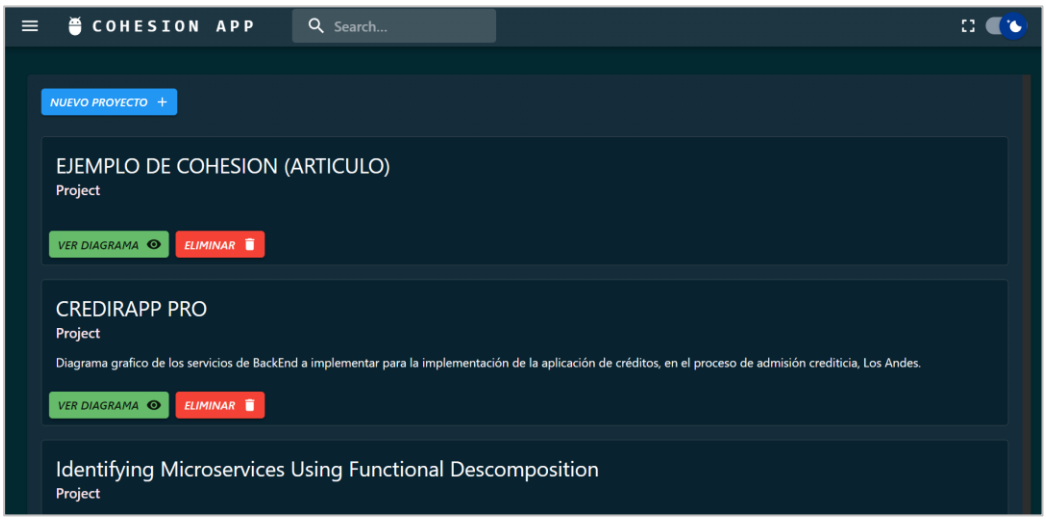
4.4.1.3. Resultados de la implementación de la aplicación web

A continuación, se presentan los resultados obtenidos después del diseño y desarrollo de la aplicación "*Diseñador de Objetos e Interrelaciones*". Cada

interfaz de esta aplicación desempeñó un papel crucial al contribuir a la gestión, creación, evaluación y visualización de proyectos y sus componentes. En esta sección, se exploraron detalladamente cada interfaz, resaltando sus funciones clave y su contribución al proceso completo de evaluación de cohesión en sistemas. Este análisis proporcionó una comprensión clara de cómo el Diseñador de objetos e interrelaciones se destacó como una herramienta significativa en el ámbito de la evaluación de Microservicios.

Figura 19

Interfaz para la gestión de proyectos.

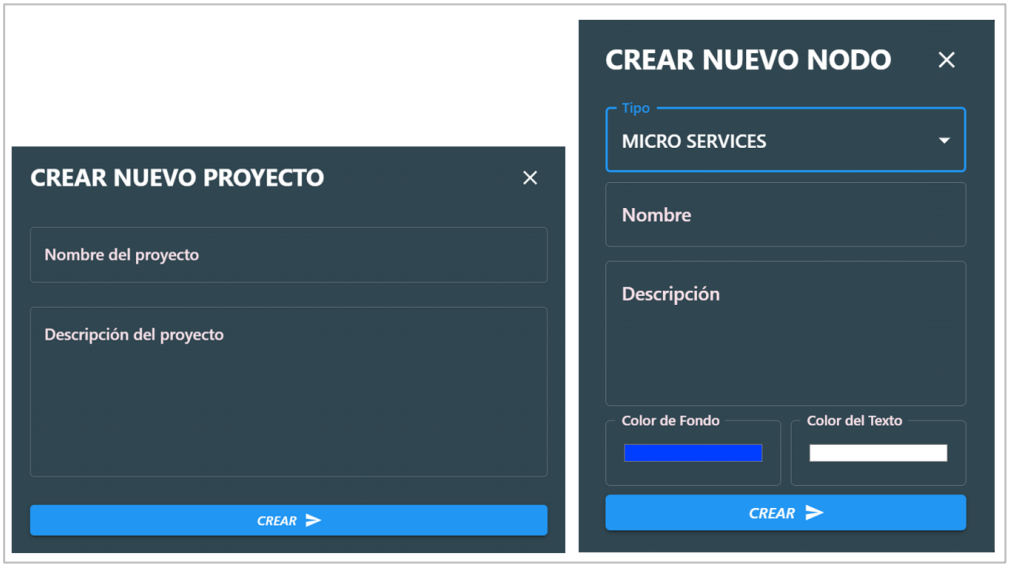


Elaboración Propia.

En la Figura 19, se muestra la interfaz dedicada a la gestión de proyectos. Esta brindaba una vista consolidada, presentando de manera accesible una lista detallada de proyectos y permitiendo, por ende, una gestión integral y flexible de diversas instancias. Se enriqueció con funcionalidades clave, como la posibilidad de añadir nuevos proyectos, eliminar los existentes y la opción de visualizar de inmediato el diagrama vinculado a cada proyecto.

Figura 20

Interfaz para la creación de proyectos y nodos.



Elaboración Propia.

En la Figura 20, se exhiben dos ventanas modales. El primer modal presenta un formulario para la creación de nuevos proyectos, recopilando información esencial como el nombre y la descripción del proyecto. El segundo modal estuvo diseñado para la creación de nodos, permitiendo especificar detalles clave como el nombre, la descripción y el tipo de nodo.

Figura 21

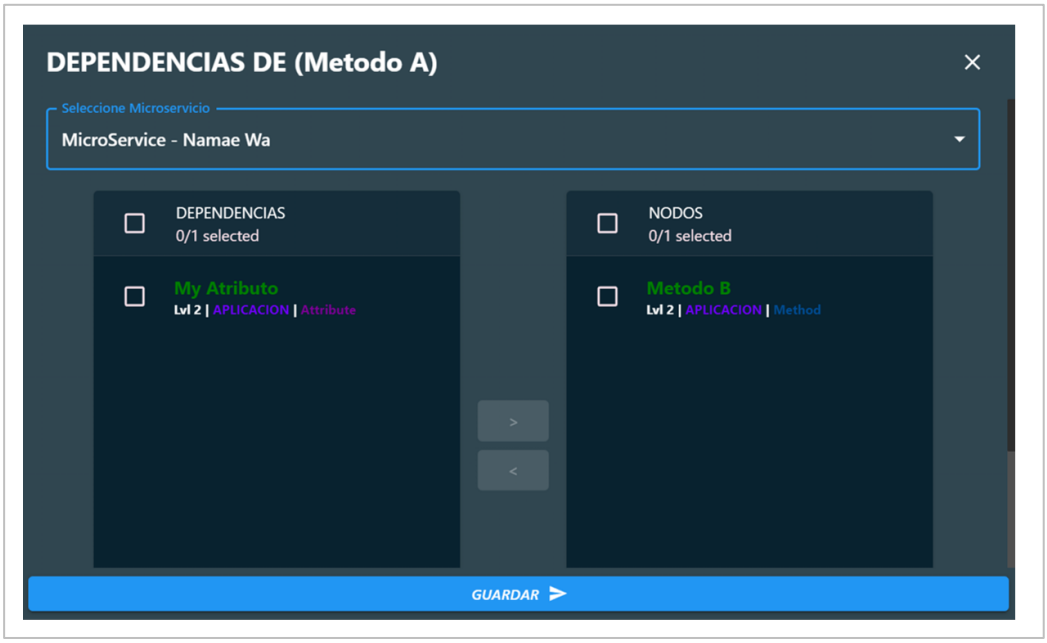
Opciones del nodo.

Elaboración Propia.

Prosiguiendo, en la Figura 21 se presentan las opciones del nodo. Esta interfaz exhibió las funciones clave asociadas a un nodo específico, abarcando la visualización de nodos hijos, la relación por dependencia con otros nodos, la evaluación de la cohesión dentro del nodo y la capacidad de eliminarlo. Estas opciones proporcionaron a los usuarios la capacidad de comprender y gestionar de manera efectiva la estructura y relaciones de cada nodo.

Figura 22

Interfaz para la relación por dependencia entre nodos.

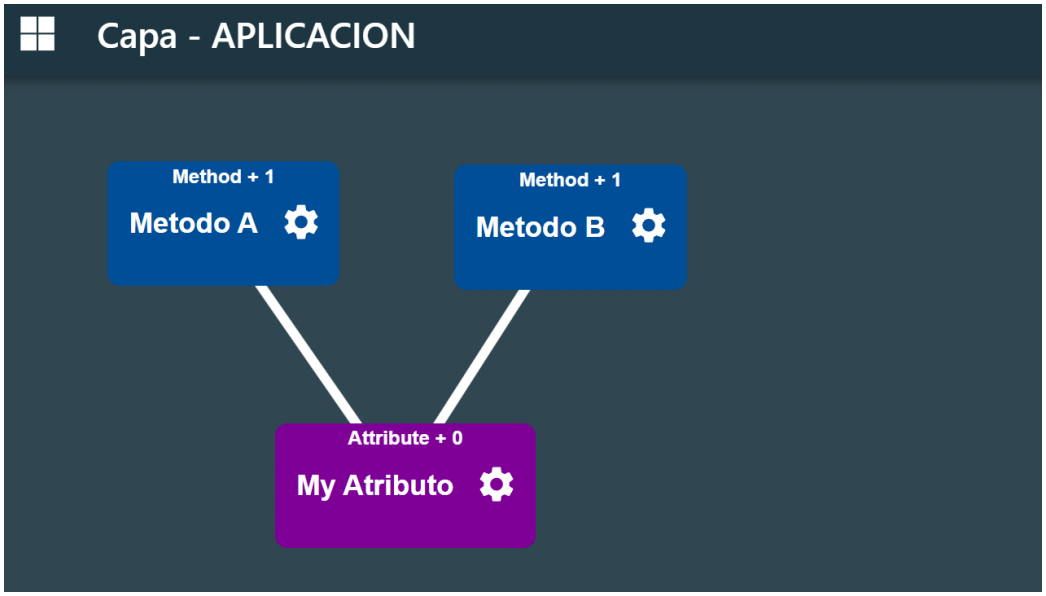


Elaboración Propia.

En la Figura 22, luego de seleccionar la opción de relacionar en el nodo, se desplegó un modal que enumeró los nodos a los cuales dependía en el "*lado izquierdo*" y los nodos disponibles para la relación en el "*lado derecho*". Esto proporcionó una representación clara y organizada de las dependencias entre nodos, permitiendo a los usuarios establecer y gestionar relaciones de dependencia de manera eficiente.

Figura 23

Representación visual de dependencias.



Elaboración Propia.

En la interfaz siguiente, mostrada en la Figura 23, se presentó una visualización gráfica detallada de los nodos, sus interrelaciones y la cantidad de dependencias por nodo en la parte superior. A través de gráficos y diagramas visuales, se logró comprender de manera sencilla la complejidad de las interrelaciones entre los componentes.

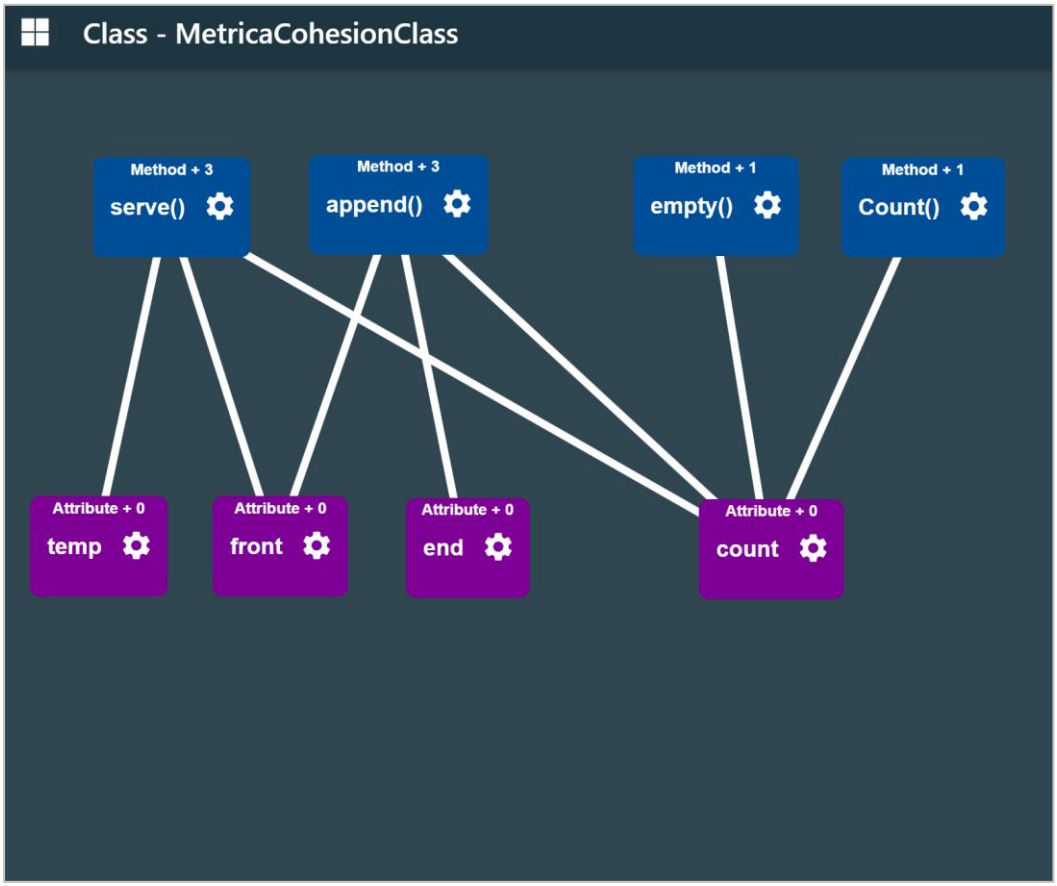
4.4.1.4. Evaluación de la métrica de cohesión de clases

A continuación, se procedió a validar la integración de la métrica de Cohesión en la aplicación web para certificar su implementación. En este proceso, se replicó la estructura de objetos presentada en el artículo de Saadati & Motameni (2014). En la Figura 24, se exhiben cuatro métodos y cuatro atributos interconectados mediante líneas blancas, siguiendo la misma configuración que se describe en el artículo de referencia. La evaluación de este conjunto de objetos se llevó a cabo conforme a los pasos detallados en el artículo original. Los resultados, expuestos en la Figura 25, revelaron una destacada cohesión de clase,

siendo notable la cohesión de la clase queue con un puntaje de 0.786. Este resultado es coincidente con el obtenido en el ejemplo del artículo de referencia, y respalda la implementación de la métrica en nuestra aplicación web. Este resultado confirma la coherencia y robustez de la métrica integrada en nuestra herramienta, proporcionando una evaluación confiable y consistente con los estándares propuestos en el marco teórico de Saadati & Motameni (2014).

Figura 24

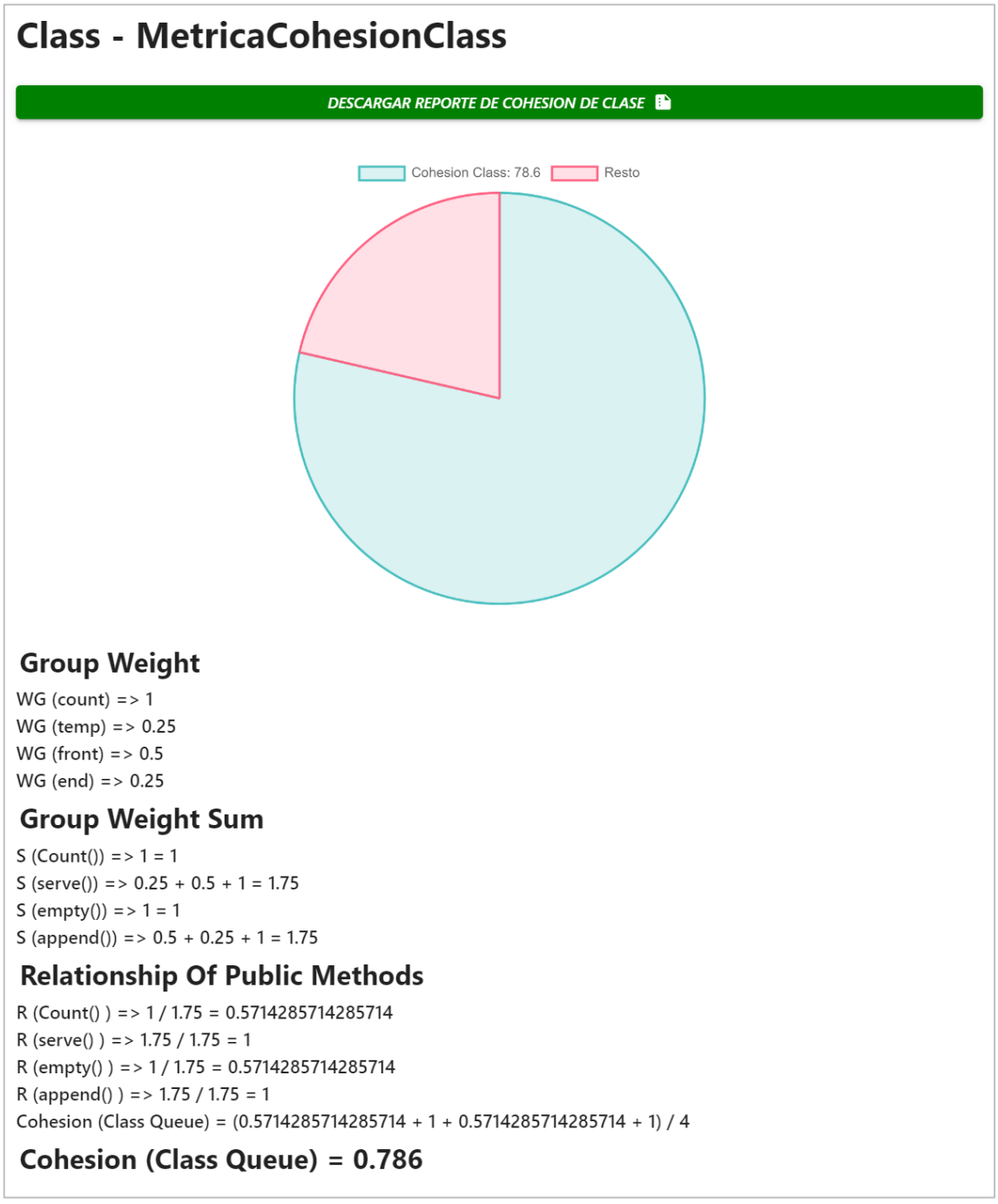
Ejemplo de clase compuesta por métodos y atributos.



Elaboración Propia.

Figura 25

Resultados del ejemplo de cohesión planteado por Saadati & Motameni.



Elaboración Propia.

4.4.2. Evaluación de la cohesión en los microservicios de Credirapp

En el siguiente apartado, se analizan los resultados derivados de la evaluación de cohesión utilizando la métrica propuesta mediante la herramienta desarrollada "*Diseñador de objetos e interrelaciones*", detallado en la Tabla 9. Este análisis proporciona una visión detallada de la interrelación y la cooperación efectiva entre los elementos del sistema, resaltando la eficacia del proceso de descomposición funcional implementado. La aplicación meticulosa de la métrica permite la evaluación de la calidad y eficacia de cada Microservicio, evaluando la cohesión interna y su impacto en el rendimiento global del sistema. Para este procedimiento, inicialmente al contar con la segmentación de los servicios previamente identificados, se llevó a cabo la creación de las clases correspondientes que albergarían la funcionalidad en cada proyecto. Estas clases están conformadas por métodos y atributos; los métodos están organizados de manera que sigan el principio de responsabilidad única, mientras que los atributos hacen referencia a conjuntos de datos que podrían provenir de tablas o segmentos de datos específicos con objetivos similares.

Tabla 9

Resumen de resultados de la cohesión de clase.

Microservicio	Porcentaje de cohesión	Acoplamiento
Clientes	**79.42%**	**Muy Bajo**
Créditos	**67.2%**	**Bajo**
Simulador	**81.25%**	**Muy Bajo**
Riesgo Crediticio	**91.65%**	**Muy Bajo**
Externos	**100%**	**Muy Bajo**
Archivos	**100%**	**Muy Bajo**

Catálogos	**100%**	**Muy Bajo**
TOTAL	**88.5%**	**Muy Bajo**

Elaboración Propia.

Interpretación: La Tabla 9, resume los resultados derivados de la evaluación de cohesión de clase realizada en cada Microservicio identificado, denotando un promedio total del 88.5%, calificándose como Muy Alta. También se aborda el acoplamiento individual de cada elemento, siendo esta entre Bajo y Muy Bajo. La relación inversamente proporcional entre la cohesión y el acoplamiento es evidente: a mayor cohesión, menor acoplamiento. Esto se debe a que cuando los elementos dentro de un módulo están altamente relacionados "*alta cohesión*" y hay pocas dependencias externas entre módulos "*bajo acoplamiento*", el sistema tiende a ser más flexible y mantenible. Esta característica permitió una adaptación eficiente a cambios y evoluciones.

4.4.2.1. Proceso de evaluación de microservicios

A continuación, se contextualiza el procedimiento empleado para la evaluación de cohesión a través de la aplicación "*Diseñador de Objetos e Interrelaciones*". En esta visualización, se representan los componentes Backend mediante nodos, los cuales son objetos programáticos interrelacionados según las necesidades del sistema. En la Figura 26, se han generado siete nodos que representan los siete Microservicios identificados. Con el fin de ilustrar este proceso, se eligió el Microservicio asociado a clientes como ejemplo representativo.

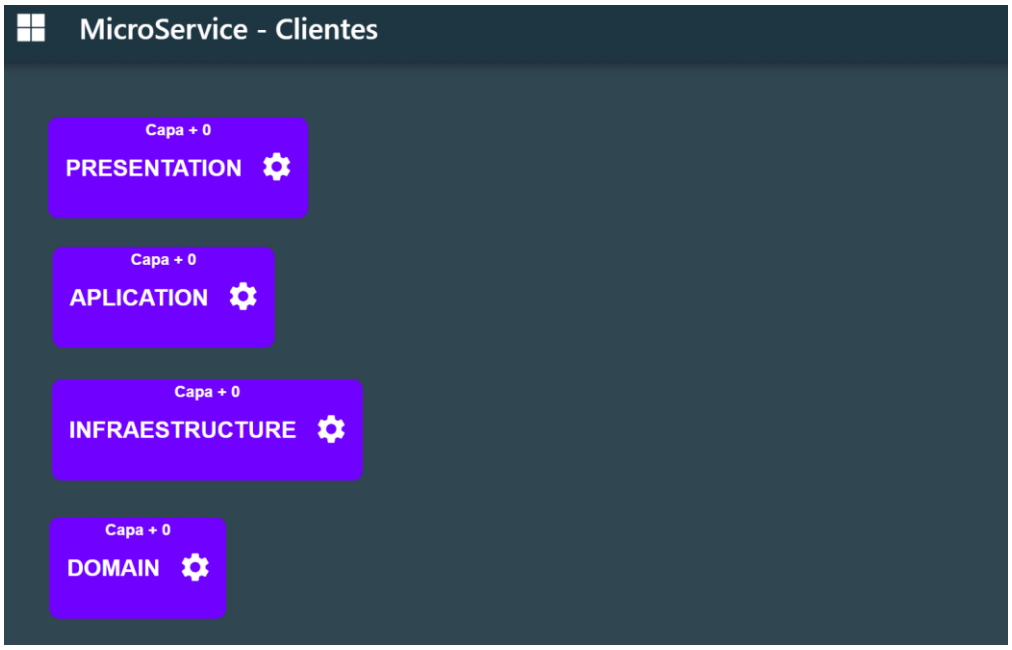
Figura 26

Representación de los siete Microservicios generados en la aplicación.

Elaboración Propia.

Figura 27

Arquitectura de Capas del Microservicio "Clientes".



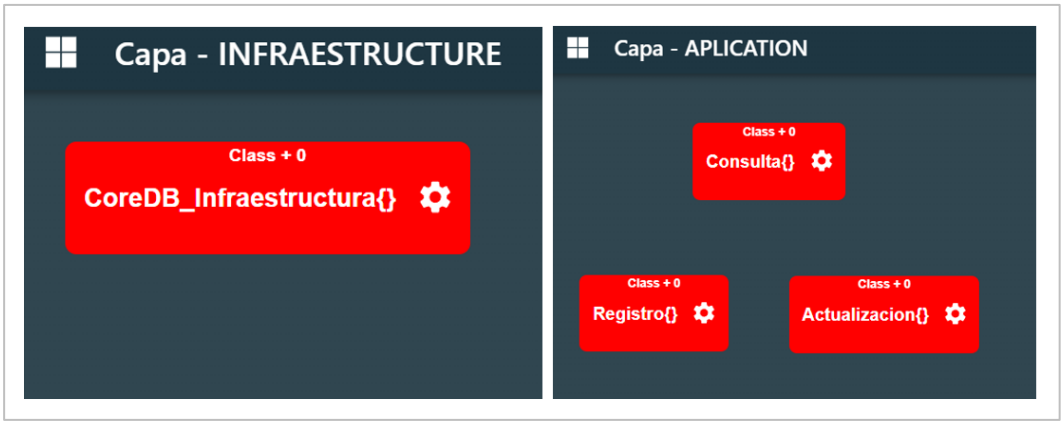
Elaboración Propia.

De acuerdo con la implementación realizada, cada Microservicio integró una arquitectura tipo "*Onion Architecture*", tal como se muestra en la Figura 27.

Esta arquitectura se compone por las capas de presentación, que estructuraba los Endpoints; aplicación, que orquestaba los bloques de datos; infraestructura, encargada de obtener los datos; y, finalmente, la capa de dominio.

Figura 28

Clases de las Capas del Microservicio "Clientes".

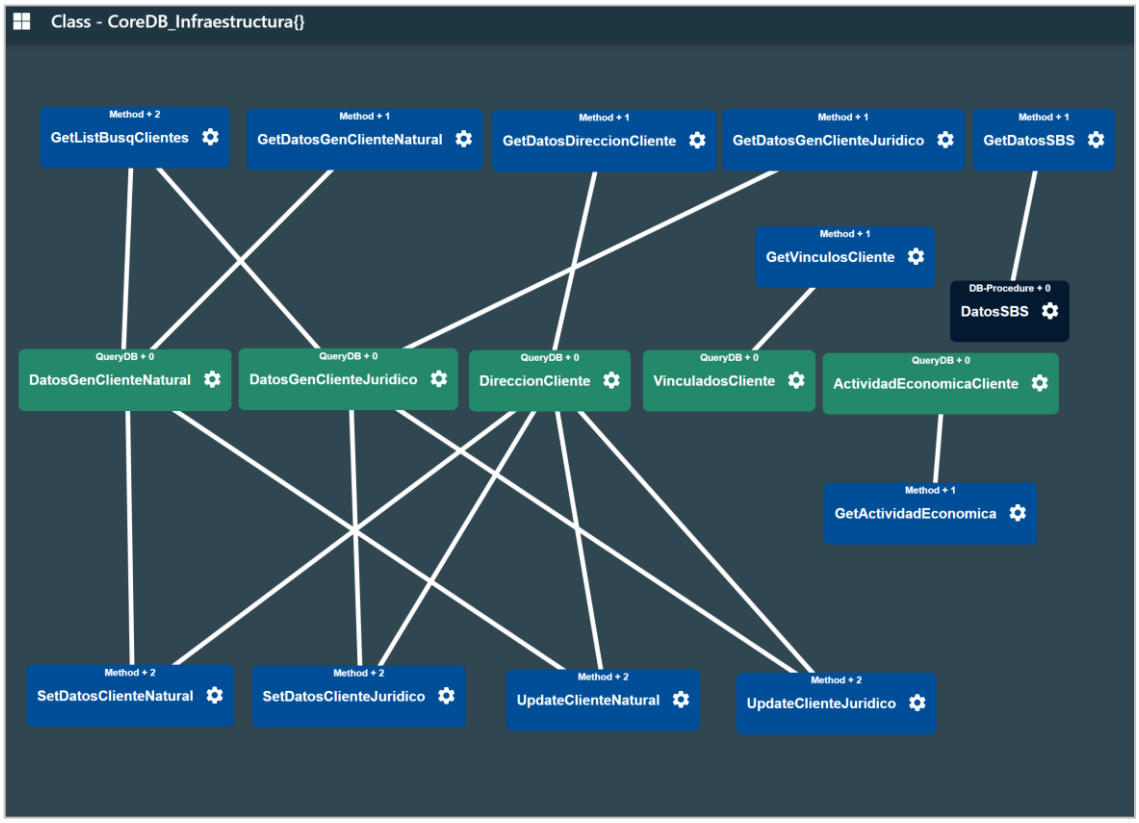


Elaboración Propia.

En la Figura 28, se presentan las clases implementadas en el Microservicio de Clientes, cada una cumpliendo funciones específicas para asegurar la coherencia y eficiencia del sistema. Se destaca la clase CoreDB_infraestructura{}, dedicada exclusivamente a la obtención de datos. En la capa de aplicación, se identificaron clases como Consulta{}, Registro{} y Actualizacion{}, diseñadas para gestionar y orquestar los bloques de datos provenientes de la infraestructura, los cuales se expondrán posteriormente a través de los Endpoints en la capa de presentación.

Figura 29

Representación de métodos y atributos del Microservicio "Clientes".



Elaboración Propia.

En la Figura 29, se visualiza la interacción entre métodos y atributos en la capa de infraestructura, específicamente en la clase CoreDB_infraestructura{}. Los métodos relacionados con lectura o escritura están representados por nodos de color azul, mientras que los atributos se muestran como grupos de datos, obtenidos mediante consultas representadas en verde. Las relaciones entre ellos se destacan mediante líneas blancas que conectan los nodos.

Esta representación ilustra que cada método actúa como una pieza de lego que luego se orquesta en la capa de aplicación. En el caso de los métodos de lectura, identificados por el prefijo "*Get*", obtienen datos de manera focalizada para mantener una alta cohesión y bajo acoplamiento. En contraste, los métodos de escritura, marcados con "*Set*" o "*Update*", al tener un propósito específico, no

requieren descomposición, ya que preservar su solidez garantiza su propósito original. Por ejemplo, el método "*SetDatosClienteNatural*" actualiza dos grupos de datos o atributos, conocidos como "*DatosGenClienteNatural*" y "*DireccionCliente*". A su vez, estos atributos se implementan en dos métodos de lectura, como "*GetDatosGenClienteNatural*" y "*GetDatosGenClienteJuridico*", desacoplados para su orquestación múltiple en la capa de aplicación. Este diseño asegura una estructura sólida y coherente en la gestión de datos del Microservicio.

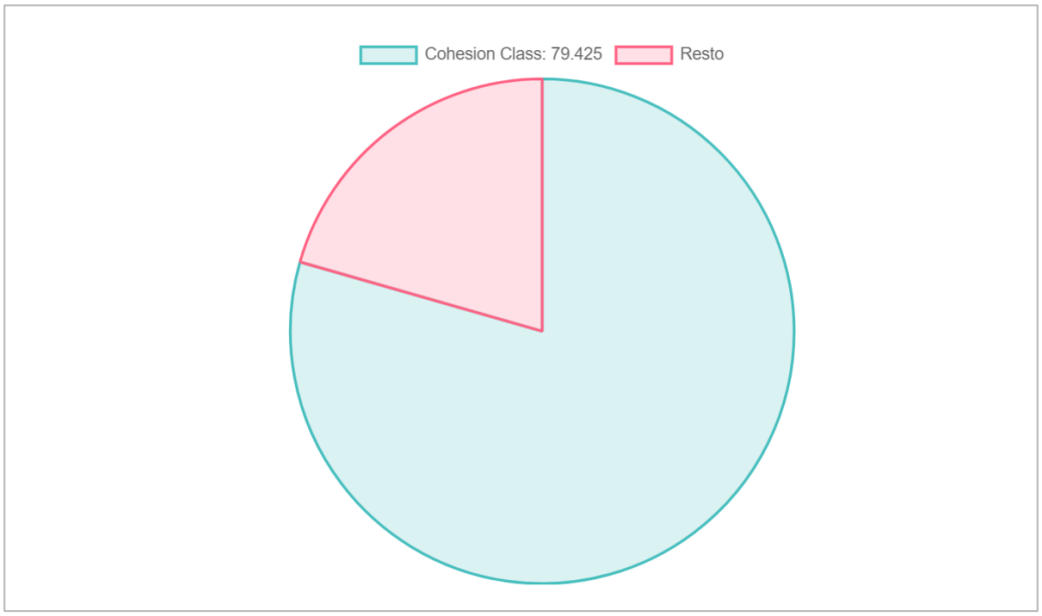
4.4.2.2. Resultados tras la evaluación de cohesión

- **Microservicio de Clientes**

El microservicio "*Clientes*" obtuvo una cohesión 79.425%, Figura 30, fue diseñado con el objetivo central de gestionar de manera eficaz las operaciones relacionadas exclusivamente con los clientes. Su enfoque se centra en funciones clave, como la consulta, registro y actualización de clientes naturales y jurídicos, así como la búsqueda de estos. Esta delimitación precisa garantiza que el microservicio mantenga una alta cohesión, ya que todas las funciones están directamente vinculadas al manejo integral de clientes. Al limitarse a estas operaciones esenciales, el microservicio logra un Muy Bajo acoplamiento, evitando redundancias y superposiciones innecesarias. Este diseño específico no solo optimiza la modularidad del sistema, sino que también simplifica de manera significativa la capacidad de adaptación y evolución del Microservicio para el futuro, sin generar impacto en otras áreas del sistema.

Figura 30

Porcentaje de cohesión en el Microservicio de Clientes.



Elaboración Propia.

Interpretación: El Microservicio de Clientes exhibe una cohesión del 79.425%, superando la media. Este Microservicio se compone de cuatro clases principales. Tres de estas clases pertenecen a la capa de Aplicación: Actualización{} con una cohesión del 100%, Consulta{} con una cohesión del 57.1%, y Registro{} con una cohesión del 100%. La cuarta clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 60.6%. El promedio de cohesión entre todas las clases es del 79.425%.

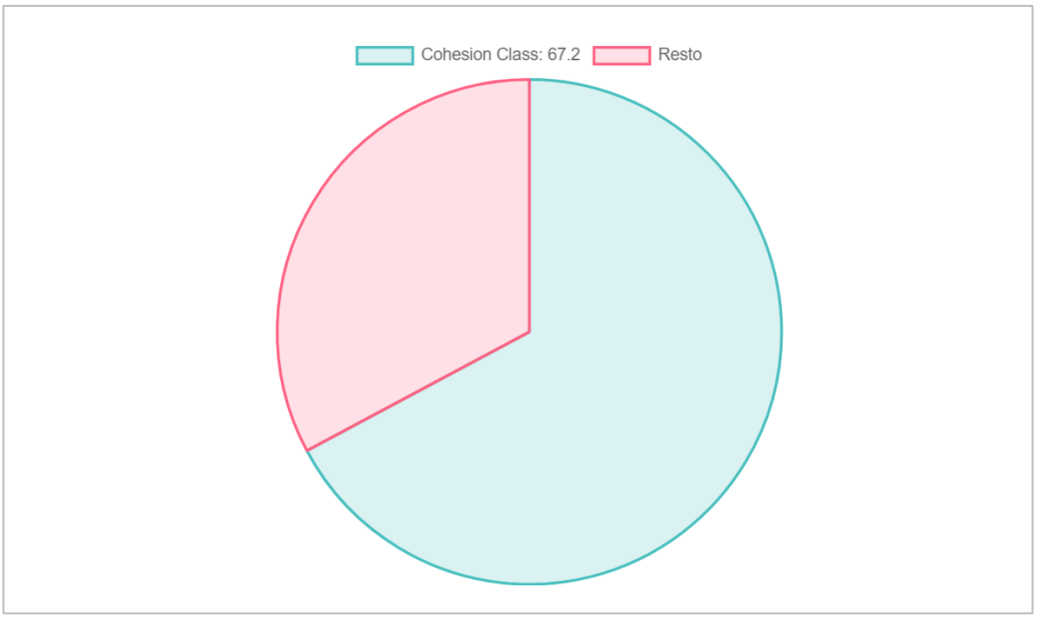
- **Microservicio de Créditos**

Este Microservicio fue diseñado con un enfoque preciso en la gestión de operaciones esenciales relacionadas exclusivamente con créditos, obtuvo una cohesión 67.2%, Figura 31. Entre sus funciones destacan la consulta de las tasas de campaña, las garantías asociadas a clientes y sus cónyuges, la revisión de las últimas evaluaciones crediticias, el registro de solicitudes de crédito, y la consulta

de la posición consolidada. Al delimitarse exclusivamente a estas tareas específicas, el Microservicio garantiza una cohesión destacada en la gestión integral asociada a los créditos.

Figura 31

Porcentaje de cohesión en el Microservicio de Créditos.



Elaboración Propia.

Interpretación: El Microservicio de Créditos exhibe una cohesión del 67.2%, superando la media. Este Microservicio se compone de tres clases principales. Dos de estas clases pertenecen a la capa de Aplicación: Registro {} con una cohesión del 100%, Consulta {} con una cohesión del 40.0%. La tercera clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 61.6%. El promedio de cohesión entre todas las clases es del 67.2%.

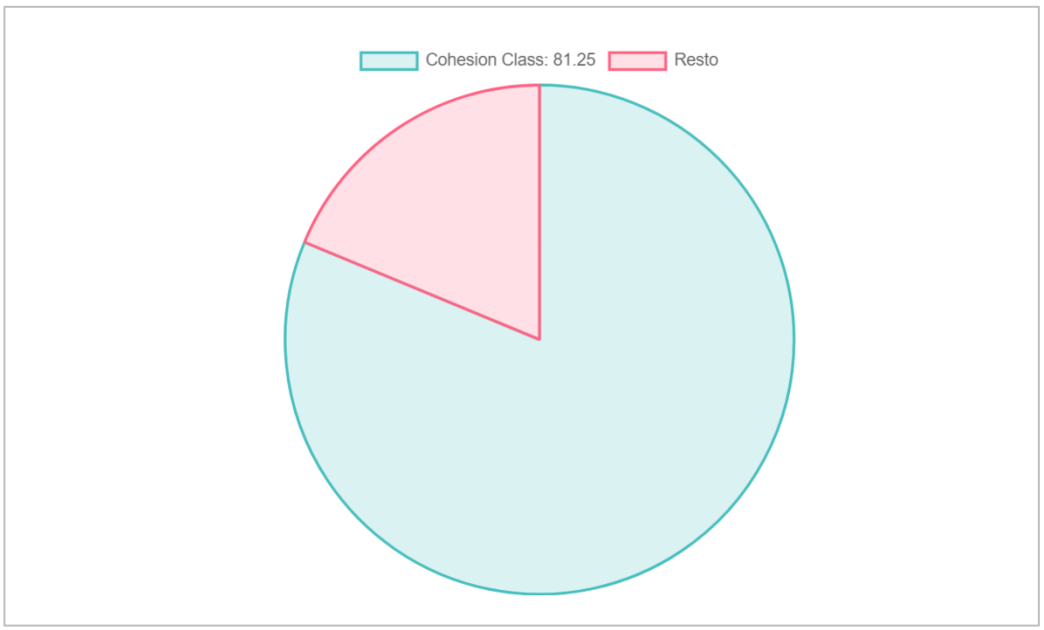
- **Microservicio de Simulador**

El Microservicio "*Simulador*" obtuvo una cohesión 81.25%, Figura 32, se especializa de manera específica en todas las funciones de simulación,

manteniendo una implementación agnóstica entre ellas. En esta se implementó el "*Simulador de Plan de Pagos*". Esta precisión en el enfoque garantiza una destacada cohesión al centrarse únicamente en la tarea fundamental de simulación. La naturaleza agnóstica del Microservicio no solo previene redundancias, sino que también simplifica ajustes y futuras mejoras sin afectar su capacidad para interoperar con otros componentes del sistema.

Figura 32

Porcentaje de cohesión en el Microservicio de Simulador.



Elaboración Propia.

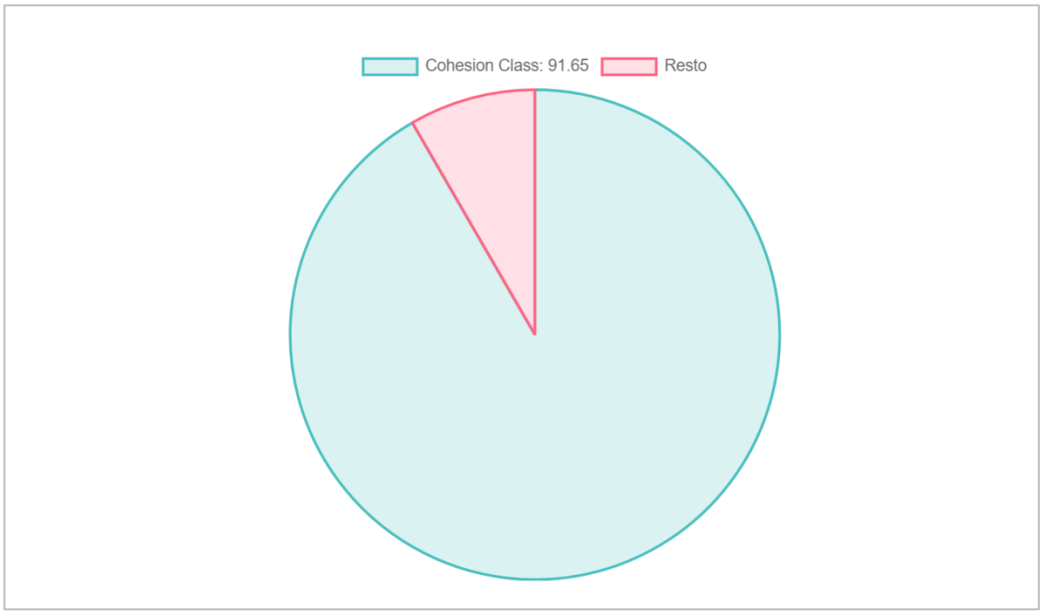
Interpretación: El Microservicio de Simulador exhibe una cohesión del 81.25%, superando la media. Este Microservicio se compone de dos clases principales. Una de estas clases pertenecen a la capa de Aplicación: Consulta{} con una cohesión del 62.5%. La segunda clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 100.0%. El promedio de cohesión entre todas las clases es del 81.25%.

- **Microservicio de Riesgo Crediticio**

Este Microservicio se centra de manera específica en el dominio de riesgos, obtuvo una cohesión 91.65%, Figura 33. En este servicio se implementa la "*Consulta sobre Endeudamiento*". Esta focalización precisa asegura que el Microservicio conserve una cohesión destacada al centrarse exclusivamente en la tarea fundamental de evaluar el nivel de riesgo asociado con el cliente. La función principal de esta consiste en suministrar información vital sobre la situación financiera de los clientes, facilitando una evaluación precisa de los riesgos crediticios asociados. Al enfocarse exclusivamente en esta operación central, el Microservicio alcanza un acoplamiento reducido, evitando redundancias y garantizando una integración eficaz con otros elementos del sistema. Esta estructura particular no solo optimiza la modularidad del sistema, sino que también simplifica ajustes y futuras actualizaciones del Microservicio sin impactar otras áreas del sistema, garantizando su eficiencia en el ámbito integral de la gestión crediticia del sistema.

Figura 33

Porcentaje de cohesión en el Microservicio de Riesgo Crediticio.



Elaboración Propia.

Interpretación: El Microservicio de Riesgo Crediticio exhibe una cohesión del 91.65%, superando la media. Este Microservicio se compone de dos clases principales. Una de estas clases pertenecen a la capa de Aplicación: Consulta{} con una cohesión del 100.0%. La segunda clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 83.3%. El promedio de cohesión entre todas las clases es del 91.65%.

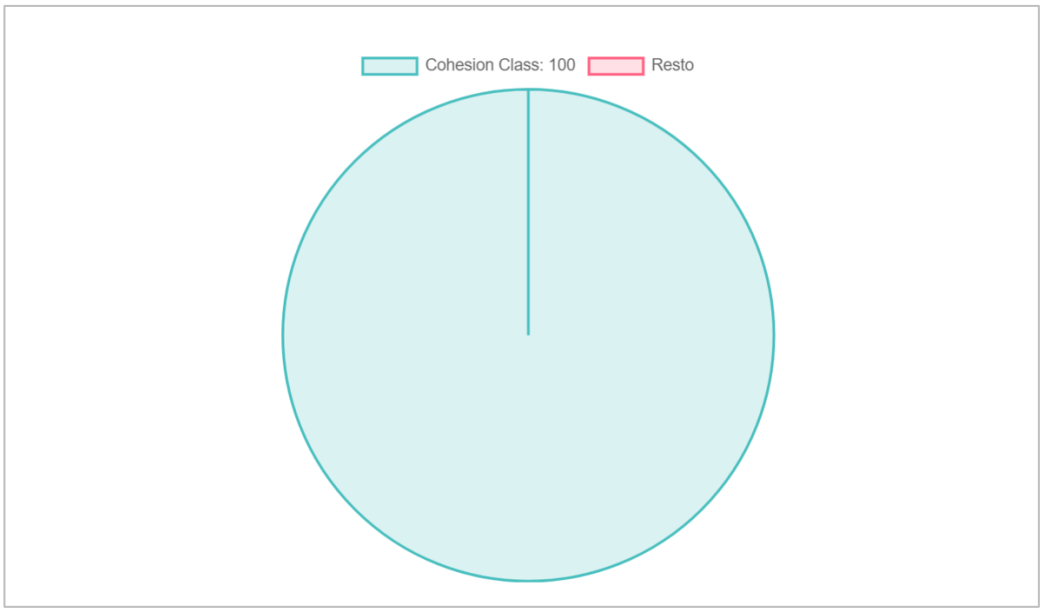
- **Microservicio de Externos**

El Microservicio "*Externos*" obtuvo una cohesión 100.00%, Figura 34, se centra en la obtención de datos externos, destacando funciones fundamentales actuales como "*Consultar SUNAT*" y "*Consultar RENIEC*". Además de estas operaciones, el Microservicio tiene la capacidad de ampliarse para obtener datos de otros proveedores. La habilidad para adquirir información de una variedad de proveedores contribuye a obtener una perspectiva integral y actualizada de la

información esencial para el sistema. Al concentrarse en estas tareas centrales, el Microservicio logra un acoplamiento reducido, evitando duplicidades y garantizando una integración eficaz con otros elementos del sistema.

Figura 34

Porcentaje de cohesión en el Microservicio de Externos.



Elaboración Propia.

Interpretación: El Microservicio de Externos exhibe una cohesión del 100.0%, lo cual denota una alta cohesión. Este Microservicio se compone de dos clases principales. Una de estas clases pertenecen a la capa de Aplicación: Consulta{} con una cohesión del 100.0%. La segunda clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 100.0%. El promedio de cohesión entre todas las clases es del 100.0%.

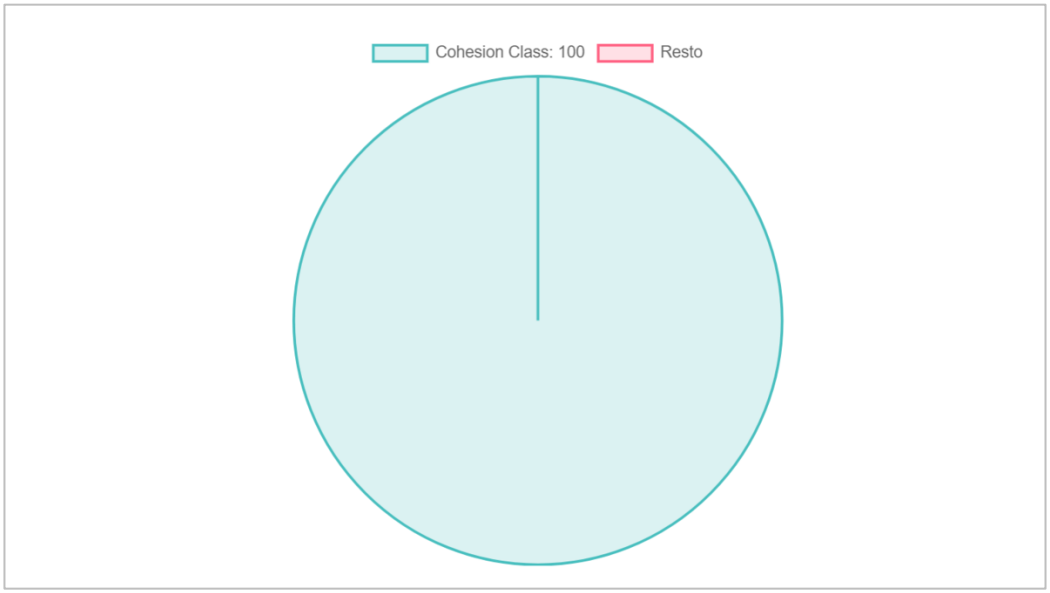
- **Microservicio de Archivos**

Este Microservicio se enfoca de manera específica en la función esencial de "*Cargar documentos, expedientes y documentos relacionados con un cliente o solicitud*" obtuvo una cohesión 100.00%, Figura 35. Este enfoque preciso asegura

que el Microservicio mantenga una alta cohesión al concentrarse exclusivamente en la tarea fundamental de gestionar archivos asociados a clientes o solicitudes. La función de carga de documentos permite una gestión eficiente de expedientes y archivos relevantes. Al delimitarse a esta operación central, el Microservicio logra un Muy Bajo acoplamiento, evitando redundancias y asegurando una integración fluida con otros componentes del sistema. Esta configuración particular simplifica ajustes y futuros desarrollos del Microservicio sin impactar otras áreas del sistema, asegurando su eficacia en el marco integral del sistema global de gestión de archivos y documentos.

Figura 35

Porcentaje de cohesión en el Microservicio de Archivos.



Elaboración Propia.

Interpretación: El Microservicio de Archivos exhibe una cohesión del 100.0%, lo cual denota una alta cohesión. Este Microservicio se compone de dos clases principales. Una de estas clases pertenecen a la capa de Aplicación: Registrar{} con una cohesión del 100.0%. La segunda clase, perteneciente a la

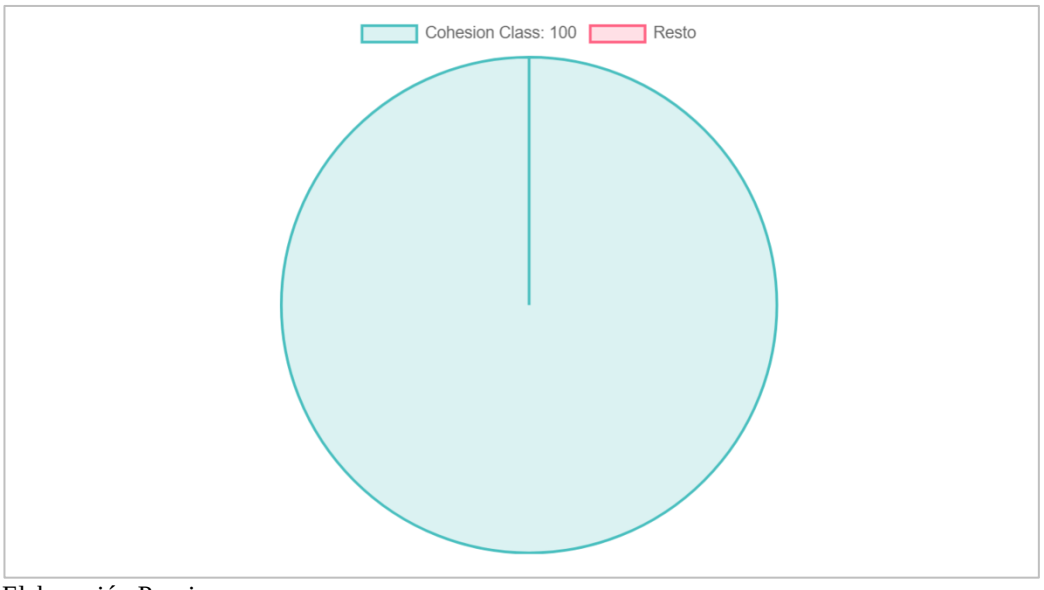
capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 100.0%. El promedio de cohesión entre todas las clases es del 100.0%.

- **Microservicio de Catálogos**

Este microservicio tiene como foco la administración de catálogos, obtuvo una cohesión 100.00%, Figura 36; estos son datos esenciales para definir colecciones utilizadas en el Frontend. Su función principal, denominada "*Actualizar y Obtener Catálogos*", asegura la constante actualización y disponibilidad de estas opciones en el Frontend. Al enfocarse en esta tarea esencial y crucial, el microservicio se distingue por su efectividad en la gestión clara y eficiente de las opciones ofrecidas al usuario dentro del marco global del sistema.

Figura 36

Porcentaje de cohesión en el Microservicio de Catálogos.



Elaboración Propia.

Interpretación: El Microservicio de Catálogos exhibe una cohesión del 100.0%, lo cual denota una alta cohesión. Este Microservicio se compone de dos clases principales. Una de estas clases pertenecen a la capa de Aplicación:

Consulta{} con una cohesión del 100.0%. La segunda clase, perteneciente a la capa de Infraestructura, es CoreDB_Infraestructura{} con una cohesión del 100.0%. El promedio de cohesión entre todas las clases es del 100.0%.

DISCUSIÓN

En el marco de la entidad financiera Los Andes, esta investigación representó un notable esfuerzo dirigido a elevar la calidad y eficiencia de los Microservicios Backend implementados. La utilización de la métrica propuesta por Saadati & Motameni (2014) reveló una estructura eficaz, destacando una cohesión general del 88.5%, clasificada como "*Muy Alta*". En paralelo, la investigación de Orjuela Velandia (2022) titulada "*Descomposición de componentes Frontend de tipo web mediante estrategias de desacoplamiento en arquitecturas de Microservicios*", abordó la descomposición de componentes Frontend en el marco de arquitecturas de Microservicios. A través de la implementación de estrategias de desacoplamiento y la aplicación de la métrica de Saadati & Motameni (2014), se logró una cohesión del 55.8%.

La métrica de Saadati & Motameni (2014), empleada en ambas tesis, demostró ser una herramienta efectiva para evaluar la cohesión en sistemas basados en Microservicios. Su aplicación meticulosa proporcionó una visión detallada de la calidad y eficacia de los componentes, permitiendo una comprensión clara de las interrelaciones y la cooperación efectiva entre ellos. Ambas investigaciones, a pesar de centrarse en aspectos distintos (Backend y Frontend), convergieron en la importancia de la cohesión para asegurar sistemas robustos y eficientes. La cohesión, evaluada a través de la métrica de Saadati & Motameni (2014), se presentó como un indicador clave para la toma de decisiones arquitectónicas y el diseño efectivo de sistemas basados en Microservicios.

V. CONCLUSIONES

PRIMERA: Se ha logrado exitosamente la aplicación de estrategias de desacoplamiento en los componentes de Backend del aplicativo Credirapp, lo que resultó en la identificación exitosa de siete Microservicios y teniendo una evaluación de la cohesión general del 88.5%, calificándose como Muy Alta. La identificación de estos Microservicios se llevó a cabo de manera efectiva mediante descomposición funcional, a partir de las necesidades del sistema expresadas en casos de uso, permitiendo conformar y segmentar grupos con fuerte cohesión de forma satisfactoria, indicando su idoneidad como Microservicios para la aplicación Credirapp. Estos siete Microservicios fueron evaluados a través de la métrica de cohesión propuesta por Saadati y Motameni, la cual se fundamenta en un árbol de subconjuntos. Los resultados de esta evaluación revelaron una cohesión Muy Alta, contribuyendo así al logro de nuestro objetivo de investigación.

SEGUNDA: Se han identificado siete Microservicios clave aplicando el método de descomposición funcional para la evaluación de la cohesión de los componentes Backend del sistema Credirapp en la entidad financiera Los Andes. Este enfoque detallado, respaldado por la asignación ponderada de operaciones de lectura y escritura, permitió un análisis preciso de las interacciones entre procesos funcionales y grupos de datos, estableciendo un fundamento sólido para la identificación de Microservicios. Además, la representación gráfica de interrelaciones demostró su eficacia al segmentar conjuntos densamente relacionados, destacando la cohesión interna y el

potencial de cada conjunto para ser un microservicio. La identificación de estos Microservicios destacó no solo por la aplicabilidad práctica del método, sino también por su relevancia estratégica en la creación de arquitecturas de Microservicios cohesivas y adaptables en entornos financieros complejos. Esto proporcionó una base sólida para el diseño de sistemas más eficientes y modulares.

TERCERA: Se modeló una arquitectura Backend basada en Microservicios para lograr una arquitectura modular, escalable y eficiente con la implementación de la arquitectura Onion alineada con los principios SOLID en el sistema Credirapp, que demostró ser una opción sólida, garantizando coherencia, mantenibilidad y adaptabilidad ante las dinámicas demandas del mercado y las regulaciones en el ámbito financiero. Cada capa cumplió de manera efectiva con su función específica, desde la encapsulación de la lógica de negocio en la capa de Dominio hasta la facilitación de la interacción externa en la capa de Presentación. La independencia y desacoplamiento entre las capas proporcionaron agilidad para enfrentar los cambios normativos en el sector financiero, permitieron una evolución fluida de los servicios y posibilitaron la rápida incorporación de nuevas funcionalidades. La metáfora de la capa de Infraestructura como un conjunto de piezas de lego resaltó la flexibilidad y adaptabilidad, siendo esencial en la eficiente gestión de bases de datos y en la reducción del acoplamiento en la administración de dependencias.

CUARTA: Se ha logrado con éxito la evaluación de cohesión en los componentes resultantes del Backend, abarcando los siete Microservicios previamente identificados, y obteniendo una cohesión general del 88.5%, calificada

como Muy Alta. La evaluación se llevó a cabo a través de la aplicación web "*Diseñador de objetos e interrelaciones*", la cual integra la capacidad para modelar, analizar y evaluar la cohesión en entornos de Microservicios. De manera satisfactoria, la evaluación de cohesión se implementó según lo establece la métrica de propuesta por Saadati y Motameni. Esta aplicación web demostró eficacia al automatizar la evaluación de la cohesión entre las clases de los Microservicios del sistema Credirapp. Esto permitió identificar la implementación del sistema en términos de cohesión, desempeñando un papel crucial en la mejora tanto de la arquitectura como de la eficiencia operativa en el desarrollo de sistemas para la entidad financiera.

VI. RECOMENDACIONES

PRIMERA: Se recomienda a la financiera Los Andes establecer un proceso continuo de revisión y mejora de Microservicios en sus actuales y futuras aplicaciones, como Credirapp, para identificar oportunidades de desacoplamiento y explorar innovaciones tecnológicas que impulsen la eficiencia de los sistemas y que permita la agilidad en la adaptación para mantener la robustez y asegurar su alineación con las cambiantes demandas del entorno financiero.

SEGUNDA: Se sugiere definir de manera clara las funcionalidades del sistema e identificar los conjuntos de datos involucrados, con el objetivo de llevar a cabo una descomposición funcional efectiva.

TERCERA: Se recomienda explorar metodologías y métricas innovadoras destinadas a mejorar las implementaciones actuales en arquitecturas de Microservicios. Además, se alienta a difundir los resultados a través de publicaciones en revistas especializadas. Este enfoque no solo fortalecerá la base de conocimientos existente, sino que también contribuirá de manera significativa a la evolución continua de la evaluación de cohesión. La aplicación posterior de estos conocimientos en la industria del software puede proporcionar beneficios tangibles. Esta recomendación busca no solo avanzar en el conocimiento académico, sino también impactar positivamente en la práctica y evolución de la evaluación de cohesión en contextos de Microservicios.

CUARTA: Finalmente, se recomienda utilizar la herramienta desarrollada "*Diseñador de objetos e interrelaciones*" para evaluar la cohesión de clases en

aplicaciones basadas en Microservicios. Su implementación mejora la precisión del análisis y representa una contribución esencial para el progreso de la investigación.

VII. REFERENCIAS BIBLIOGRÁFICAS

Addison-Wesley. (2012). *Software Systems Architecture Working With Stakeholders Using Viewpoints And Perspectives*.

Anubha Sharma, Manoj Kumar, & Sonali Agarwal. (2015). *A Complete Survey on Software Architectural Styles and Patterns*.

Aral, A., & Ovatman, T. (2013). Utilization of Method Graphs to Measure Cohesion in Object Oriented Software. *Annual Computer Software and Applications Conference Workshops*, 505-510. https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/

Barnes, J. M., Garlan, D., & Schmerl, B. (2012). *Evolution styles foundations and models for software architecture evolution*.

Burgos Díaz, S. (2021). *DESTRUCCIÓN DE UN MONOLITO EN MICROSERVICIOS EN FANDANGO LATINOAMÉRICA*.

Christopher Newman, Zach G. Agioutantis, & Nathaniel Schaefer. (2018). *Development of a web-platform for mining applications*.

Gamma, Erich. (2002). *Patrones de diseño : elementos de software orientado a objetos reutilizable*. Addison Wesley.

Godbolt, M. (2016). *Frontend Architecture for Design Systems A MODERN BLUEPRINT FOR SCALABLE AND SUSTAINABLE WEBSITES*. www.it-ebooks.info

Hall, G. McLean. (2014). *Adaptive code via C# : Agile coding with design patterns and SOLID principles*. Microsoft Press.

Harms, H., Rogowski, C., & Lo Iacomo, L. (2017). *Guidelines for adopting frontend architectures and patterns in microservices-based systems*. https://www.researchgate.net/publication/318872312_Guidelines_for_adopting_frontend_architectures_and_patterns_in_microservices-based_systems

Harrison, R., Counsell, S. J., & Nithi, R. V. (1998). *An evaluation of the MOOD set of object-oriented software metrics*.

J. McC Smith, & D. Stotts. (2002). *Elemental design patterns a formal semantics for composition of OO software architecture*.

Jha, P. C., Bali, V., Narula, S., & Kalra, M. (2014). Optimal component selection based on cohesion & coupling for component based software system under build-or-buy scheme. *Journal of Computational Science*, 233-242.

Kniberg, H. (2015). *SCRUM AND XP FROM THE TRENCHES*.

Lewis, J., & Fowler, M. (2014). *Microservices a definition of this new architectural term*. martinFowler.com. https://www.martinfowler.com/articles/microservices.html

López Hinojosa, J. D. (2017). *ARQUITECTURA DE SOFTWARE BASADA EN MICROSERVICIOS PARA DESARROLLO DE APLICACIONES WEB DE LA ASAMBLEA NACIONAL*. Universidad Técnica del Norte.

Lung, C.-H., Xu, X., & Zaman, M. (2007). *SOFTWARE ARCHITECTURE DECOMPOSITION USING ATTRIBUTES*.

Machuca Pajuelo, M. O. (2021). *Implementación de la práctica DevSecOps aplicada a microservicios para una entidad financiera*. Universidad Nacional Mayor de San Marcos.

Martin, R. C. (2009). *Código limpio Manual de estilo para el desarrollo ágil de software*.

Martin, R. C. (2017). *Clean Architecture: a Craftsman's Guide to Software Structure and Design*.

Martin, R. C., & Kriens, P. (2012). *Java Application Architecture Modularity Patterns With Examples Using OSGi*.

Martin, R. C., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*.

Mitchell, B. S., & Mancoridis, S. (2006). *On the Automatic Modularization of Software Systems Using the Bunch Tool*.

Moreno Bernal, S. D. (2022). *Modelo de Evolución Arquitectónica de Monolito a Microservicios para el Sistema de Información ISys de la Dirección Nacional de*

Admisiones de la Universidad Nacional de Colombia. Universidad Nacional de Colombia.

Nebel, A. (2018). *Arquitectura de Microservicios para Plataformas de Integración*. Universidad de la República Montevideo.

Newman, S. (2015). *Building Microservices Designing Fine-Grained Systems*.

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* (Inc. " "O'Reilly Media, Ed.).

Nieto Sánchez, J. (2017). *Modelo de Arquitectura de Software para Aplicaciones iOS basado en Clean Architecture*.

Orjuela Velandia, C. C. (2022). *Descomposición de componentes front-end de tipo web mediante estrategias de desacoplamiento en arquitecturas de microservicios*. Universidad Nacional de Colombia.

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, & Judith Stafford. (2010). *Documenting Software Architectures: Views and Beyond*.

Pedraza Coello, R. (2021). *Método DISC: Separando sistemas en microservicios*. Universidad Nacional Autónoma de México.

Portal ISO/IEC 25000. (2023). https://www.iso.org/obp/ui/es/#iso:std:iso-iec:25000:ed-2:v1:en

Pressman, R. S. (2010). *Ingeniería del software : un enfoque práctico*. McGraw-Hill.

Richard N. Taylor, Nenad Medvidovic, & Eric Dashofy. (2009). *Software Architecture: Foundations, Theory, and Practice*.

Richards, M. (2022). *Software Architecture Patterns, 2nd Edition*.

Richardson, C. (2014). *Pattern: Monolithic Architecture*. https://microservices.io/patterns/monolithic.html

Richardson, C. (2018). *Microservices Patterns With examples in Java*.

Richardson, C., & Smith, F. E. (2016). *Microservices From Design to Deployment*.

Rivera, V., & Humberto, F. (2021). *Modelo inteligente de especificación de la granularidad de aplicaciones basadas en microservicios*.

Rozanski, N., & Woods, E. (s. f.). *Software systems architecture: working with stakeholders using viewpoints and perspectives*.

S. Mahmood 1, R. Lai 1, & Y.S. Kim 2. (2007). *Survey of component-based software development*.

Saadati, M., & Motameni, H. (2014). Measuring Cohesion And Coupling Of Object-oriented Systems. *Journal of Mathematics and Computer Science*, *09*(02), 149-156. https://doi.org/10.22436/jmcs.09.02.08

Taylor, J. (2023). *Clean Architecture in ASP.NET Core*. https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/

Tyszberowicz, S., Heinrich, R., Liu, B., & Liu, Z. (2018). *Identifying Microservices Using Functional Decomposition*.

ANEXOS

ANEXO 1: Procedimiento para la aplicación de la métrica de cohesión

MICROSERVICIO DE CLIENTES		
CAPA	NOMBRE DE LA CLASE	COHESIÓN
Infraestructura	CoreDB_Infraestructura{}	0.606
Aplicación	Consulta{}	0.571
Aplicación	Registro{}	1
Aplicación	Actualizacion{}	1
COHESIÓN DEL MICROSERVICIO: (0.606 + 0.571 + 1 + 1) / 4 = 0.79425		

CLASE: CoreDB_Infraestructura{}	
Total de métodos (TM)	11
Total de métodos en subconjuntos	5
Pesos de grupos (WG)	WG: DatosGenClienteNatural = 0.8 WG: VinculadosCliente = 0.2 WG: DatosGenClienteJuridico = 0.8 WG: DireccionCliente = 1 WG: DatosSBS = 0.2 WG: ActividadEconomicaCliente = 0.2

Suma de peso de grupos (SM)	S: GetDatosGenClienteNatural => 0.8 = 0.8 S: GetVinculosCliente => 0.2 = 0.2 S: UpdateClienteJuridico => 0.8 + 1 = 1.8 S: GetDatosGenClienteJuridico => 0.8 = 0.8 S: SetDatosClienteNatural => 0.8 + 1 = 1.8 S: SetDatosClienteJuridico => 0.8 + 1 = 1.8 S: GetDatosSBS => 0.2 = 0.2 S: GetActividadEconomica => 0.2 = 0.2 S: UpdateClienteNatural => 0.8 + 1 = 1.8 S: GetListBusqClientes => 0.8 + 0.8 = 1.6 S: GetDatosDireccionCliente => 1 = 1
Máximo S(M) del árbol	1.8
Relación de los métodos públicos (RM)	R (GetDatosGenClienteNatural) => 0.8 / 1.8 = 0.4444444444444445 R (GetVinculosCliente) => 0.2 / 1.8 = 0.11111111111111112 R (UpdateClienteJuridico) => 1.8 / 1.8 = 1 R (GetDatosGenClienteJuridico) => 0.8 / 1.8 = 0.4444444444444445 R (SetDatosClienteNatural) => 1.8 / 1.8 = 1 R (SetDatosClienteJuridico) => 1.8 / 1.8 = 1 R (GetDatosSBS) => 0.2 / 1.8 = 0.11111111111111112 R (GetActividadEconomica) => 0.2 / 1.8 = 0.11111111111111112 R (UpdateClienteNatural) => 1.8 / 1.8 = 1

	R (GetListBusqClientes) => 1.6 / 1.8 = 0.888888888888889 R (GetDatosDireccionCliente) => 1 / 1.8 = 0.5555555555555556
Ecuación para la Cohesión de clase	(0.4444444444444445 + 0.11111111111111112 + 1 + 0.4444444444444445 + 1 + 1 + 0.11111111111111112 + 0.11111111111111112 + 1 + 0.888888888888889 + 0.5555555555555556) / 11
Cohesión de clase	0.606

CLASE: Consulta{}	
Total de métodos (TM)	4
Total de métodos en subconjuntos	2
Pesos de grupos (WG)	WG: GetVinculosCliente = 0.5 WG: GetListBusqClientes = 0.5 WG: GetActividadEconomica = 1 WG: GetDatosGenClienteNatural = 0.5 WG: GetDatosDireccionCliente = 1 WG: GetDatosSBS = 1 WG: GetDatosGenClienteJuridico = 0.5
Suma de peso de grupos (SM)	S: ConsultaVinculados 0.5 = 0.5 S: BusquedaCliente 0.5 = 0.5 S: ConsultaDatosCliente 1 + 0.5 + 1 + 1 = 3.5

	S: ConsultaJuridicos 1 + 1 + 0.5 + 1 = 3.5
Máximo S(M) del árbol	3.5
Relación de los métodos públicos (RM)	R (ConsultaVinculados) => 0.5 / 3.5 = 0.14285714285714285 R (BusquedaCliente) => 0.5 / 3.5 = 0.14285714285714285 R (ConsultaDatosCliente) => 3.5 / 3.5 = 1 R (ConsultaJuridicos) => 3.5 / 3.5 = 1
Ecuación para la Cohesión de clase	(0.14285714285714285 + 0.14285714285714285 + 1 + 1) / 4
Cohesión de clase	0.571

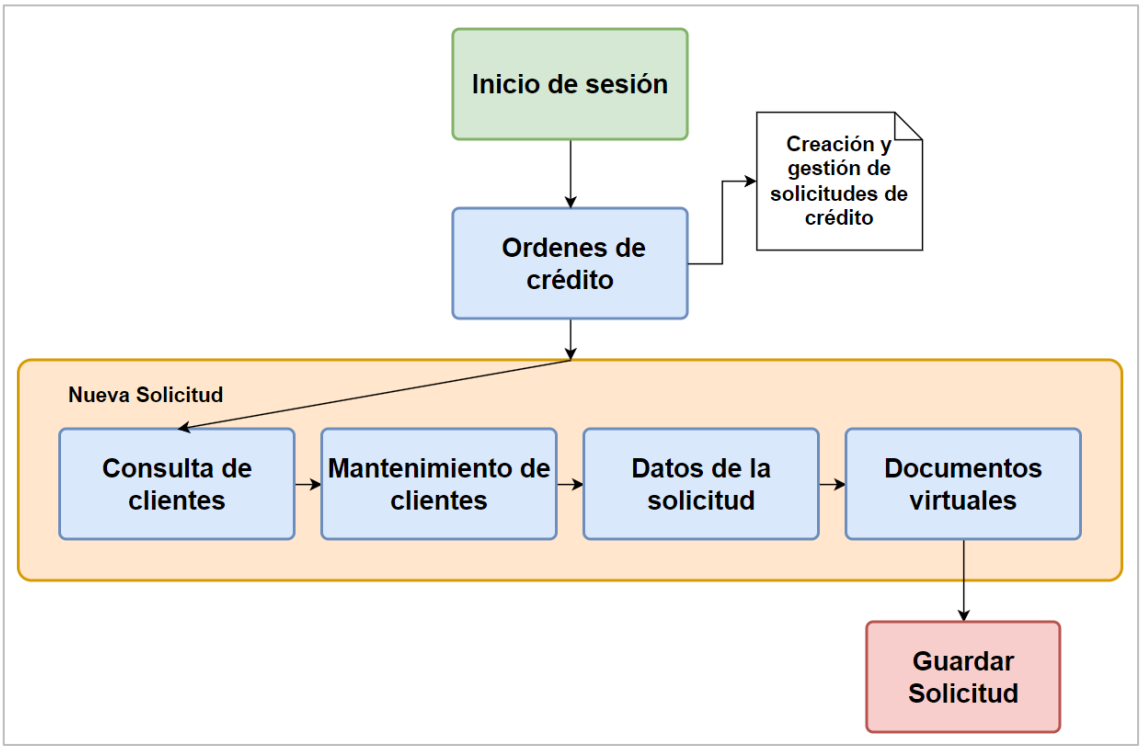
CLASE: Registro{}	
Total de métodos (TM)	2
Total de métodos en subconjuntos	1
Pesos de grupos (WG)	WG: SetDatosClienteNatural = 1 WG: SetDatosClienteJuridico = 1
Suma de peso de grupos (SM)	S: RegistraClienteNatural 1 = 1 S: RegistraClienteJuridico 1 = 1
Máximo S(M) del árbol	1
Relación de los métodos públicos (RM)	R (RegistraClienteNatural) => 1 / 1 = 1 R (RegistraClienteJuridico) => 1 / 1 = 1

Ecuación para la Cohesión de clase	(1 + 1) / 2
Cohesión de clase	1

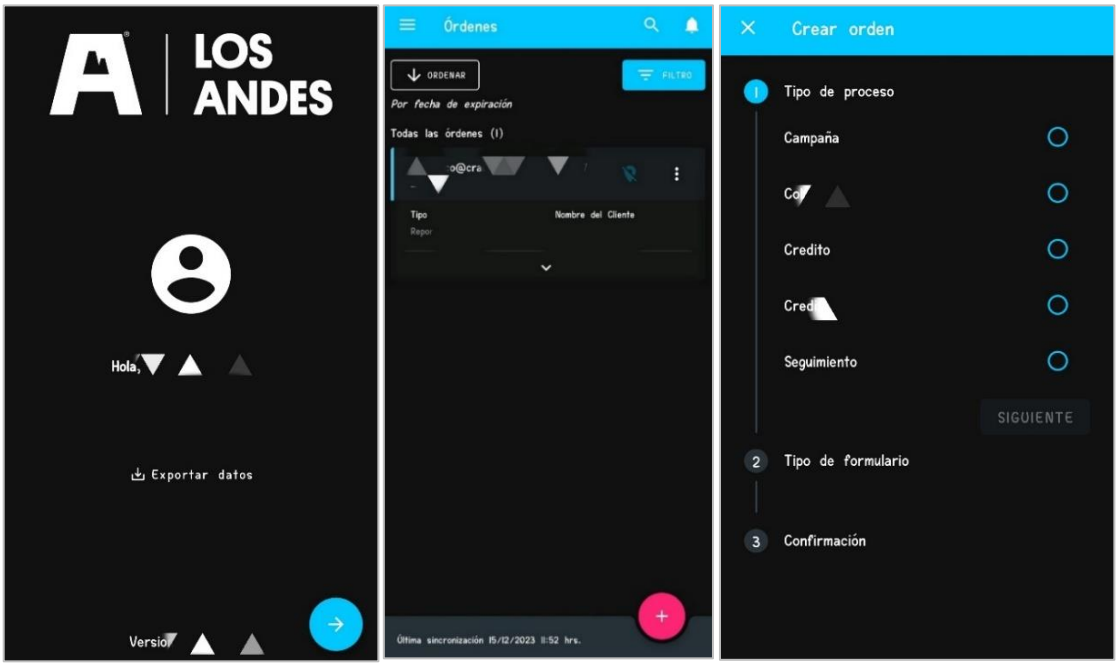
CLASE: Actualizacion{}

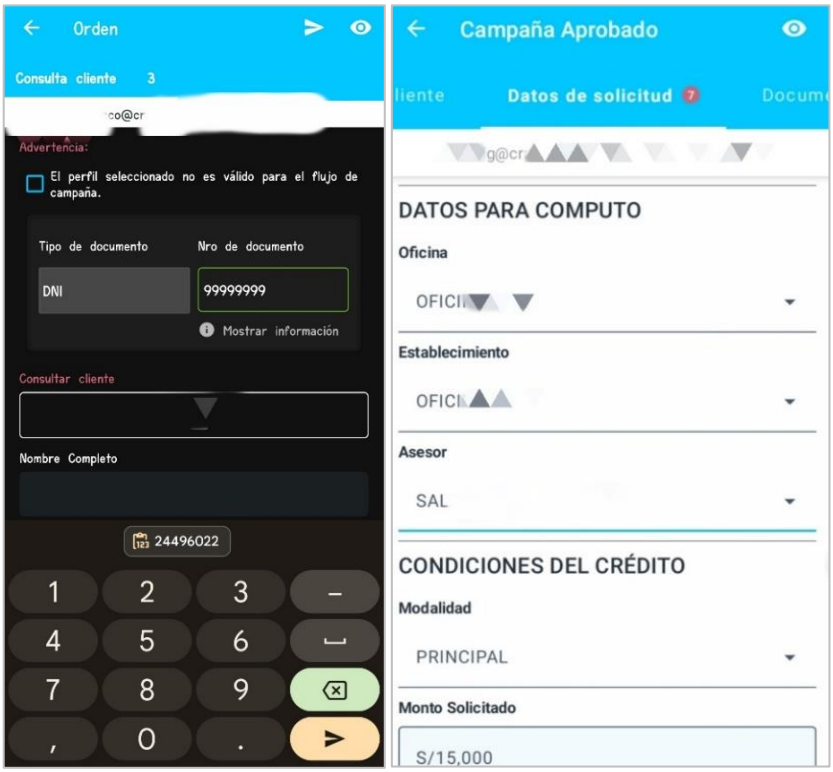
Total de métodos (TM)	2
Total de métodos en subconjuntos	1
Pesos de grupos (WG)	WG: UpdateClienteJuridico = 1 WG: UpdateClienteNatural = 1
Suma de peso de grupos (SM)	S: ActualizaClienteJuridico 1 = 1 S: ActualizaClienteNatural 1 = 1
Máximo S(M) del árbol	1
Relación de los métodos públicos (RM)	R (ActualizaClienteJuridico) => 1 / 1 = 1 R (ActualizaClienteNatural) => 1 / 1 = 1
Ecuación para la Cohesión de clase	(1 + 1) / 2
Cohesión de clase	1

ANEXO 2: Diagrama de navegación de la aplicación Credirapp



ANEXO 3: Capturas de pantalla de la aplicación móvil Credirapp





ANEXO 4: Declaración jurada de autenticidad de tesis

Universidad Nacional del Altiplano Puno | VRI Vicerrectorado de Investigación | Repositorio Institucional

DECLARACIÓN JURADA DE AUTENTICIDAD DE TESIS

Por el presente documento, Yo **Alex Fredy Escalante Maron**, identificado con DNI N° **73104785** en mi condición de egresado de la **Escuela Profesional de Ingeniería de Sistemas**, informo que he elaborado la **Tesis** denominada: **"EVALUACIÓN DE LA COHESIÓN ENTRE COMPONENTES BACKEND BASADOS EN MICROSERVICIOS MEDIANTE LA APLICACIÓN DE ESTRATEGIAS DE DESACOPLAMIENTO EN LA ENTIDAD FINANCIERA LOS ANDES"**.

Es un tema original.

Declaro que el presente trabajo de tesis es elaborado por mi persona y **no existe plagio/copia** de ninguna naturaleza, en especial de otro documento de investigación (tesis, revista, texto, congreso, o similar) presentado por persona natural o jurídica alguna ante instituciones académicas, profesionales, de investigación o similares, en el país o en el extranjero.

Dejo constancia que las citas de otros autores han sido debidamente identificadas en el trabajo de investigación, por lo que no asumiré como suyas las opiniones vertidas por terceros, ya sea de fuentes encontradas en medios escritos, digitales o Internet.

Asimismo, ratifico que soy plenamente consciente de todo el contenido de la tesis y asumo la responsabilidad de cualquier error u omisión en el documento, así como de las connotaciones éticas y legales involucradas.

En caso de incumplimiento de esta declaración, me someto a las disposiciones legales vigentes y a las sanciones correspondientes de igual forma me someto a las sanciones establecidas en las Directivas y otras normas internas, así como las que me alcancen del Código Civil y Normas Legales conexas por el incumplimiento del presente compromiso

Puno 22 de enero del 2024

Alex Fredy Escalante Maron
DNI N° 73104785

Huella

ANEXO 5: Autorización para el depósito de tesis en el Repositorio Institucional

Universidad Nacional del Altiplano Puno | VRI Vicerrectorado de Investigación | Repositorio Institucional

AUTORIZACIÓN PARA EL DEPÓSITO DE TESIS O TRABAJO DE INVESTIGACIÓN EN EL REPOSITORIO INSTITUCIONAL

Por el presente documento, Yo **Alex Fredy Escalante Maron**, identificado con DNI N° **73104785** en mi condición de egresado de la **Escuela Profesional de Ingeniería de Sistemas**, informo que he elaborado la **Tesis** denominada: **"EVALUACIÓN DE LA COHESIÓN ENTRE COMPONENTES BACKEND BASADOS EN MICROSERVICIOS MEDIANTE LA APLICACIÓN DE ESTRATEGIAS DE DESACOPLAMIENTO EN LA ENTIDAD FINANCIERA LOS ANDES"** para la obtención de **Título Profesional.**

Por medio del presente documento, afirmo y garantizo ser el legítimo, único y exclusivo titular de todos los derechos de propiedad intelectual sobre los documentos arriba mencionados, las obras, los contenidos, los productos y/o las creaciones en general (en adelante, los "Contenidos") que serán incluidos en el repositorio institucional de la Universidad Nacional del Altiplano de Puno.

También, doy seguridad de que los contenidos entregados se encuentran libres de toda contraseña, restricción o medida tecnológica de protección, con la finalidad de permitir que se puedan leer, descargar, reproducir, distribuir, imprimir, buscar y enlazar los textos completos, sin limitación alguna.

Autorizo a la Universidad Nacional del Altiplano de Puno a publicar los Contenidos en el Repositorio Institucional y, en consecuencia, en el Repositorio Nacional Digital de Ciencia, Tecnología e Innovación de Acceso Abierto, sobre la base de lo establecido en la Ley N° 30035, sus normas reglamentarias, modificatorias, sustitutorias y conexas, y de acuerdo con las políticas de acceso abierto que la Universidad aplique en relación con sus Repositorios Institucionales. Autorizo expresamente toda consulta y uso de los Contenidos, por parte de cualquier persona, por el tiempo de duración de los derechos patrimoniales de autor y derechos conexos, a título gratuito y a nivel mundial.

En consecuencia, la Universidad tendrá la posibilidad de divulgar y difundir los Contenidos, de manera total o parcial, sin limitación alguna y sin derecho a pago de contraprestación, remuneración ni regalía alguna a favor mío; en los medios, canales y plataformas que la Universidad y/o el Estado de la República del Perú determinen, a nivel mundial, sin restricción geográfica alguna y de manera indefinida, pudiendo crear y/o extraer los metadatos sobre los Contenidos, e incluir los Contenidos en los índices y buscadores que estimen necesarios para promover su difusión.

Autorizo que los Contenidos sean puestos a disposición del público a través de la siguiente licencia:

Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional. Para ver una copia de esta licencia, visita: https://creativecommons.org/licenses/by-nc-sa/4.0/

En señal de conformidad, suscribo el presente documento.

Puno 22 de enero del 2024

Alex Fredy Escalante Maron
DNI N° 73104785

Huella