



EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación
y Desarrollo Tecnológico

Tesis de Doctorado

Método para descomponer la base de datos
de sistemas monolíticos, para operar en
arquitecturas de microservicios

presentada por

M.C. Omar Hernández López

como requisito para la obtención del
grado de

**Doctor en Ciencias de la
Computación**

Director de tesis

Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Septiembre de 2026.

	ACEPTACIÓN DE IMPRESIÓN DEL DOCUMENTO DE TESIS DOCTORAL	Código: CENIDET-AC-006-D20
	Referencia a la Norma ISO 9001:2008 7.1, 7.2.1, 7.5.1, 7.6, 8.1, 8.2.4	Revisión: 0
		Página 1 de 1

Cuernavaca, Mor., a 10 de septiembre de 2026

DR. CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
P R E S E N T E

AT'n: **DRA. ANDREA MAGADÁN SALAZAR**
PRESIDENTA DEL CLAUSTRO DOCTORAL DEL
DEPARTAMENTO DE CIENCIAS COMPUTACIONALES

Los abajo firmantes, miembros del Comité Tutorial del estudiante **M.C. OMAR HERNÁNDEZ LÓPEZ** manifiestan que después de haber revisado el documento de tesis titulado **"Método para descomponer la base de datos de sistemas monolíticos, para operar en arquitecturas de microservicios"**, realizado bajo la dirección del **Dr. René Santaolaya Salgado**, el trabajo se ACEPTA para proceder a su impresión.

ATENTAMENTE

Excelencia en Educación Tecnológica®
"Conocimiento y Tecnología al Servicio de México"


DR. RENÉ SANTAOLAYA SALGADO
TECNM/CENIDET


DR. JUAN CARLOS ROJAS PÉREZ
TECNM/CENIDET


DRA. BLANCA DINA VALENZUELA ROBLES
TECNM/CENIDET


DR. NOÉ ALEJANDRO CASTRO SÁNCHEZ
TECNM/CENIDET


DR. SERGIO FABIAN RUIZ PAZ
UNIVERSIDAD DEL PAPALOAPAN

c.c.p: Verónica Sotelo Boyas /Jefa del Departamento de Servicios Escolares
c.c.p: Noé Alejandro Castro Sánchez / Jefe del Departamento de Ciencias Computacionales
c.c.p: Expediente



Educación
Secretaría de Educación Pública



TECNOLÓGICO
NACIONAL DE MÉXICO



Centro Nacional de Investigación y Desarrollo Tecnológico
Subdirección Académica

Cuernavaca Mor, 11/septiembre/2026

Oficio No. SAC/305/2026

Asunto: Autorización de impresión de tesis

OMAR HERNÁNDEZ LÓPEZ
CANDIDATO AL GRADO DE DOCTOR EN CIENCIAS
DE LA COMPUTACIÓN
PRESENTE

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"Método para descomponer la base de datos de sistemas monolíticos, para operar en arquitecturas de microservicios"**, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

Esperando que el logro del mismo sea acorde con sus aspiraciones profesionales, reciba un cordial saludo.

ATENTAMENTE

Excelencia en Educación Tecnológica

"Conocimiento y Tecnología al servicio de México"

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO



c.c.p. Departamento de Ciencias Computacionales
Departamento de Servicios Escolares

CMAZ/lmz



2026
año de
Margarita
Maza

cenidet
Centro Nacional de Investigación
y Desarrollo Tecnológico



Interior Internado Palmira S/N, Col. Palmira,
C.P. 62490, Cuernavaca, Morelos Tel. 01 (777) 3627770, ext. 4106,
e-mail: acad_cenidet@tecnm.mx | cenidet.tecnm.mx

Dedicatoria

Dedico este trabajo, en primer lugar, a mi esposa Diana y a mi hijo Mateo, quienes han sido mi mayor inspiración durante este largo camino. Gracias por su amor, comprensión, paciencia y apoyo incondicional a lo largo de mis estudios de doctorado. Prometo devolverles el tiempo prestado.

A la memoria de mi madre Oralia, quien hoy me acompaña desde el cielo. Gracias por enseñarme a nunca permanecer inmóvil ante los desafíos y oportunidades de la vida, por motivarme siempre a superarme y por inculcarme la importancia de establecer metas y trabajar con determinación para alcanzarlas. Su ejemplo, sus consejos y su cariño continúan guiando mis pasos y forman parte de cada uno de mis logros.

A mi padre Francisco, por enseñarme con su ejemplo que la constancia, la disciplina y la dedicación son los pilares fundamentales para alcanzar cualquier objetivo. Gracias por sus enseñanzas, por su esfuerzo y por demostrarme que el trabajo perseverante siempre da frutos.

Finalmente, dedico también este logro a mis hermanas Adriana y Mónica, cuya confianza y apoyo han sido la fuerza que me impulsó a continuar aun en los momentos difíciles.

Agradecimientos

En primer lugar, agradezco a Dios por permitirme culminar esta importante etapa de mi formación profesional. Gracias por brindarme salud, fortaleza física y mental, así como la perseverancia necesaria para superar los desafíos que se presentaron durante el desarrollo de mis estudios.

Expreso mi más profundo agradecimiento a mi director de tesis, el Dr. René Santaolaya Salgado, a quien considero una persona honorable y un ejemplo de profesionalismo. Gracias por su orientación, por compartir generosamente sus conocimientos y experiencias, y por aquellas charlas matutinas que, además de enriquecer esta investigación, estuvieron llenas de reflexión y aprendizaje.

A mis asesores, la Dra. Blanca Dina Valenzuela Robles, Dr. Juan Carlos Rojas Pérez, al Dr. Noé Alejandro Castro Sánchez, y al Dr. Sergio Fabián Ruiz Paz les agradezco sinceramente su tiempo, experiencia y valiosas observaciones. Sus sugerencias y conocimientos fueron fundamentales para orientar y fortalecer el desarrollo de este trabajo de investigación.

A mis compañeros y ahora amigos Antonio, Elías, Iliana, Fernando, Ninfa y Nélida, gracias por su amistad, apoyo y compañerismo durante mi estancia en el CENIDET. Compartir este camino con ustedes hizo que esta experiencia fuera más enriquecedora tanto en lo académico como en lo personal.

Finalmente, expreso mi agradecimiento a la Secretaría de Ciencia, Humanidades, Tecnología e Innovación (SECIHTI), al Tecnológico Nacional de México (TecNM), al Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) por el apoyo institucional, económico y administrativo brindado durante la realización de mis estudios de doctorado.

Resumen

La migración de sistemas monolíticos hacia arquitecturas de microservicios enfrenta un problema recurrente: la descomposición de la lógica de negocio puede realizarse sin una descomposición equivalente de la base de datos del monolito. Cuando múltiples servicios continúan accediendo a un mismo esquema de datos, se introduce un acoplamiento indirecto que limita su autonomía y dificulta su evolución y escalabilidad. En este trabajo se presenta MONO2MSDB, un método sistemático para la descomposición de bases de datos monolíticas mediante análisis léxico-sintáctico y el patrón de diseño Visitor. El método define una estrategia de análisis de dependencias que puede adaptarse, en principio, a distintos lenguajes de programación mediante gramáticas compatibles con ANTLR, mientras que su implementación actual está especializada en sistemas monolíticos desarrollados en Java. El método identifica las cadenas de dependencia entre los componentes de las capas de presentación, lógica de negocio y persistencia que participan en la ejecución de un caso de uso. En su implementación actual, dichas cadenas se establecen entre controladores REST, servicios, repositorios y entidades de persistencia basadas en Java Persistence API (JPA), a partir de las cuales se generan microservicios funcionales con modelos de datos aislados. Un conjunto de reglas identifica las entidades de datos compartidas entre casos de uso y determina su asignación a los microservicios candidatos. Para cada candidato, MONO2MSDB genera el código fuente de un microservicio y la infraestructura necesaria para sincronizar, mediante el patrón Observador, los datos de las entidades compartidas entre servicios. El método fue validado mediante la métrica Índice de Acoplamiento entre Tablas Compartidas (IATC) calculado antes y después de la migración en tres sistemas de distintos dominios y mediante una evaluación empírica que involucró 17 sistemas monolíticos provenientes de repositorios públicos. Los resultados muestran una reducción estadísticamente significativa del acoplamiento asociado a los datos compartidos, con una reducción promedio de aproximadamente 91 %, lo que indica la efectividad del enfoque propuesto para reducir las dependencias estructurales durante la migración de monolitos a microservicios. Los resultados muestran que MONO2MSDB constituye un mecanismo automatizado y reproducible para apoyar la modernización de sistemas monolíticos, al reducir el acoplamiento asociado al acceso compartido a datos y la intervención manual en la identificación de microservicios, la partición de datos y la generación de código.

Palabras clave: microservicios, descomposición de base de datos, sistemas monolíticos, análisis léxico-sintáctico, acoplamiento indirecto, modernización de software.

Abstract

The migration of monolithic systems to microservice architectures faces a recurring problem: business logic can be decomposed without an equivalent decomposition of the monolith's database. When multiple services continue to access the same data schema, indirect coupling is introduced, limiting their autonomy and hindering their evolution and scalability. This work presents MONO2MSDB, a systematic method for decomposing monolithic databases through lexical-syntactic analysis and the Visitor design pattern. The method defines a dependency analysis strategy that can, in principle, be adapted to different programming languages through ANTLR-compatible grammars, while its current implementation is specialized for monolithic systems developed in Java. The method identifies dependency chains among components of the presentation, business logic, and persistence layers involved in the execution of a use case. In its current implementation, these chains are established among REST controllers, services, repositories, and Java Persistence API (JPA)-based persistence entities, from which functional microservices with isolated data models are generated. A set of rules identifies data entities shared across use cases and determines their assignment to candidate microservices. For each candidate, MONO2MSDB generates the source code of a microservice and the infrastructure required to synchronize, through the Observer pattern, the data of entities shared among services. The method was validated using the Shared Table Coupling Index (IATC), calculated before and after migration in three systems from different domains, and through an empirical evaluation involving 17 monolithic systems obtained from public repositories. The results show a statistically significant reduction in coupling associated with shared data, with an average reduction of approximately 91%, indicating the effectiveness of the proposed approach in reducing structural dependencies during the migration of monoliths to microservices. The results show that MONO2MSDB constitutes an automated and reproducible mechanism for supporting the modernization of monolithic systems by reducing coupling associated with shared data access and manual intervention in microservice identification, data partitioning, and code generation.

Keywords: microservices, database decomposition, monolithic systems, lexical-syntactic analysis, indirect coupling, software modernization

Índice

Lista de figuras	vi
Lista de tablas	vi
Capítulo 1: Introducción.....	1
1.1 Presentación	1
Capítulo 2: Antecedentes	2
2.1 Planteamiento del problema.....	2
2.2 Justificación	2
2.2.1 Situación Causal	3
2.3 Objetivos	4
2.3.1 Objetivo general	4
2.3.2 Objetivos específicos.....	4
2.4 Alcances y limitaciones	4
2.5 Fundamentos	6
2.5.1 Marco teórico	6
2.6 Estado del arte.....	18
2.6.1 Trabajos relacionados.....	18
2.6.2 Trabajos internos del laboratorio de ingeniería de software.....	28
Capítulo 3: Materiales y Métodos	30
3.1 Descripción general del Método MONO2MSDB.....	30
3.2 Fundamentos Conceptuales del Método	31
3.2.1 Arquitectura en Capas como Precondición del Método	31
3.2.2 La Secuencia Interactiva de Casos de Uso como unidad de Descomposición	32
3.3 Fase 1: Análisis Sintáctico del Código Fuente.....	33
3.4 Fase 2: Identificación de Componentes por Estrato Arquitectónico.....	34
3.5 Fase 3: Resolución transversal en la secuencia de interacción	36
3.6 Fase 4: Resolución Transitiva de Relaciones entre Entidades del Dominio	37
3.7 Fase 5: Separación del Modelo de Datos Distribuido.....	38
3.8 Fase 6: Generación de Artefactos del Microservicio	41
3.9 Enfoque general del método.	42
3.9.1 Análisis del sistema monolítico.....	43
3.9.2. Identificación de microservicios.....	43
3.9.3 Generación de Gramática y SQL.....	43
3.9.4 Configuración del Analizador Léxico-Sintáctico	44
3.9.5 Implementación del Patrón de Diseño Visitor.....	44
3.9.6 Generación Automática de Código para Microservicios.....	44
3.10 Estructura de almacenamiento de datos	45

3.10.1 Elementos de un caso de uso.....	45
3.10.2 Elementos Específicos de Base de Datos (SQL)	47
3.11 Implementación del Método en la Herramienta Informática	48
3.11.1 Diagrama de Casos de Uso.....	49
3.11.2 Diagrama de secuencias	52
3.11.3 Diagrama de clases de la implementación de la herramienta.	53
Capítulo 4: Pruebas	54
4.1 Evaluación del método propuesto.....	54
4.2. Validación del método en herramienta informática	54
4.2.1 Alcance.....	55
4.2.2 Plan de pruebas.....	56
4.2.3 Selección de los casos de prueba.....	58
4.2.4 Diseño de pruebas	58
4.2.5 Casos de pruebas	59
4.3. Evaluación experimental del método.....	77
4.3.1 Metodología empleada	77
4.3.2 Pregunta de investigación e hipótesis.....	78
4.3.3 Variables dependientes e independientes	78
4.3.4 Diseño del plan de pruebas.....	79
4.3.5 Enfoque y Criterios de Aceptación.....	80
4.3.6 Diseño de la prueba	80
4.3.7 Caso de prueba	81
4.3.8 Aplicación de la métrica.....	84
4.4. Ejecución del experimento.....	85
4.4.1 Muestreo y estratificación	85
4.4.2 Cálculo del tamaño de la muestra.....	85
4.4.3 Asignación proporcional por estratos.....	85
4.4.4 Aplicación de la métrica IATC.....	87
4.4.5 Cálculo del vector de diferencias	88
4.4.6 Evaluación de normalidad: prueba de Shapiro–Wilk.....	88
4.4.7 Selección de la prueba de hipótesis	89
4.4.8 Prueba Wilcoxon	89
4.4.9 Tamaño del efecto (Cohen’s d pareado).....	90
4.4.10 Intervalo de confianza del 95%	91
4.4.11 Análisis de los resultados visuales.	91
4.4.12 BoxPlot comparativo entre IATC_Monolito vs IATC_Migrado	91
4.4.13 Gráficas de líneas pareada	92
4.4.14 Boxplot por estratos.....	93
4.4.15 Conclusión respecto a la hipótesis Inicial del método.....	95

4.4.16 Conclusiones comparativas entre estratos	96
Capítulo 5. Análisis e interpretación de resultados	96
Capítulo 6: Conclusiones y trabajo futuro	98
Referencias	100
Anexos.....	104
Anexos A – Conceptos generales	104
Anexos B – Sustento Teórico de la Métrica IATC.	105
Anexos C – Herramienta informática	109

Lista de figuras

Figura 1 Ejemplo de la arquitectura monolítica usando el patrón de diseño MVC.....	8
Figura 2 Principio de una base de datos por servicio en arquitecturas de microservicios.	9
Figura 3 Proceso BPMN del método para descomponer la BD de un sistema monolítico hacia microservicios.	31
Figura 4 Mapa de las relaciones entre los elementos del caso de uso.	46
Figura 5 Mapa de las relaciones de un método perteneciente a un caso de uso.	48
Figura 6 Diagrama general de casos de uso de la herramienta MONO2MSDB.	50
Figura 7 Diagrama principal de caso de uso.	52
Figura 8 Diagrama de secuencia de la herramienta MONO2MSDB.	53
Figura 9 Diagrama de clases de la herramienta informática.	54
Figura 10 Diagrama de clases del sistema de ventas.....	60
Figura 11 Diagrama de clases del sistema de facturación.....	67
Figura 12 Diagrama de clases del sistema de gestión de fidelización.....	72
Figura 13 Boxplot IATC_Monolito vs IATC_Migrado.....	92
Figura 14 Gráfico pareado por repositorio.	93
Figura 15 Reducción de IATC por tamaño del repositorio.	94

Lista de tablas

Tabla 1 Características de la Arquitectura Monolítica	8
Tabla 2 Comparación entre arquitecturas monolíticas y arquitecturas de microservicios.	10
Tabla 3 Relación entre cohesión, acoplamiento y autonomía de servicios.	11
Tabla 4 Comparación entre los enfoques de base de datos compartida y base de datos por servicio. ...	12
Tabla 5 Comparación entre técnicas de análisis de software.	13
Tabla 6 Etapas fundamentales del procesamiento de lenguajes de programación.	15
Tabla 7 Uso de ANTLR y el patrón Visitor para la obtención de secuencias de interacción.	16

Tabla 8 Resumen de artículos sobre trabajos relacionados con esta investigación.....	26
Tabla 9 Estrategias de clasificación de componentes por metadatos explícitos.....	34
Tabla 10 Reglas de identificación de secuencias interactivas de caso de uso.....	35
Tabla 11 Reglas formales de separación del modelo de datos distribuido.....	39
Tabla 12 Modelo de separación de datos para una entidad compartida E.....	40
Tabla 13 Módulos Maven de la herramienta MONO2MSDB.	49
Tabla 14 Dependencias tecnológicas de la herramienta.....	49
Tabla 15 Descripción general de los casos de uso que intervienen en la herramienta.	50
Tabla 16 Niveles y tipos de pruebas.....	56
Tabla 17 Componente del entorno de pruebas	58
Tabla 18 Generación de bases de datos por microservicio.....	58
Tabla 19 Identificar el modelo de datos	59
Tabla 20 Casos de prueba.....	59
Tabla 21 Relaciones entre las tablas del sistema de gestión de ventas.	61
Tabla 22 Relación de casos de uso con el modelo de datos del sistema de gestión de ventas.	61
Tabla 23 Generación de bases de datos por microservicio para el sistema de gestión de ventas.....	61
Tabla 24 Evaluación del funcionamiento del caso de uso CategoriaControllerTest.....	64
Tabla 25 Evaluación del funcionamiento del caso de uso ClienteControllerTest.....	65
Tabla 26 Evaluación del funcionamiento del caso de uso PedidoControllerTest.	66
Tabla 27 Relaciones entre las entidades del sistema de gestión de facturación.....	68
Tabla 28 Relación de casos de uso con el modelo de datos del sistema de gestión de facturación.	68
Tabla 29 Generación de bases de datos por microservicio para el sistema de gestión de facturación..	68
Tabla 30 Evaluación del funcionamiento del caso de uso InvoiceServiceTest.....	70
Tabla 31 Evaluación del funcionamiento del caso de uso DiscountServiceTest.	70
Tabla 32 Evaluación del funcionamiento del caso de uso ReturnServiceTest.....	71
Tabla 33 Relaciones entre las entidades del modelo	73
Tabla 34 Relaciones entre las entidades del sistema de gestión de fidelización.	73
Tabla 35 Generación de DB por microservicio para el sistema de gestión de fidelización.	73
Tabla 36 Evaluación del funcionamiento del caso de uso ReturnServiceTest.....	75
Tabla 37 Evaluación del funcionamiento del caso de uso LoyaltyServiceEnrollTest.	75
Tabla 38 Evaluación del funcionamiento del caso de uso LoyaltyServiceAddPointsTest.....	76
Tabla 39 Diseño de prueba “Reducción del acoplamiento indirecto”.....	81
Tabla 40 Repositorios divididos por estratos según sus casos de uso.....	86
Tabla 41 Datos recolectados al aplicar la métrica	87
Tabla 42 Resumen numérico por estrato.....	94

Capítulo 1: Introducción

1.1 Presentación

En las últimas décadas, los sistemas de información empresariales han evolucionado hacia arquitecturas cada vez más complejas para satisfacer las demandas de escalabilidad, disponibilidad, interoperabilidad y rapidez en la entrega de nuevas funcionalidades. Durante muchos años, la arquitectura monolítica constituyó el enfoque predominante para el desarrollo de aplicaciones empresariales debido a su simplicidad de implementación y despliegue. Sin embargo, conforme estos sistemas crecieron en tamaño y complejidad, comenzaron a presentar problemas relacionados con el mantenimiento, la evolución tecnológica, la escalabilidad y el acoplamiento entre sus componentes.

La arquitectura orientada a servicios (SOA) y el estilo arquitectónico de microservicios consideran al servicio como el principal componente para implementar funcionalidades de negocios. Las arquitecturas basadas en servicios tienen en común que son distribuidas; lo cual significa que los componentes de servicio se acceden de forma remota a través de algún tipo de protocolo de acceso remoto (por ejemplo, HTTP sobre Representational State Transfer (REST), Simple Object Access Protocol (SOAP) o Java Message Service (JMS)).

La arquitectura de microservicios apareció como un nuevo paradigma para la programación de aplicaciones, mediante la composición de sistemas integrando pequeños servicios, cada uno de los cuales ejecuta su propio proceso y se comunica con otros servicios a través de mecanismos ligeros (Fowler & Lewis, s. f.). Este enfoque se basa en los conceptos de SOA, a partir de la identificación de procesos y flujos de trabajo de la organización que atienden a una única responsabilidad o meta de valor.

La forma natural y evolutiva ha llevado a los sistemas monolíticos a migrar hacia nuevas tecnologías y arquitecturas como lo son el cómputo en la nube, la arquitectura orientada a servicios (SOA) y los microservicios (MS). Estos últimos tienen ventajas significativas ya que permiten mayor capacidad de almacenamiento, mayor tolerancia a fallas, son escalables y quizá la ventaja más importante es el aislamiento. Esto hace que la arquitectura orientada a microservicios (MSA) sea una alternativa al pensar en migrar software monolítico heredado.

A pesar de la creciente adopción de este paradigma, una gran cantidad de organizaciones continúan operando sistemas monolíticos desarrollados durante años o incluso décadas. La migración de estos sistemas hacia arquitecturas de microservicios representa un desafío significativo debido a la complejidad de identificar límites adecuados de servicio, preservar la lógica de negocio existente y, particularmente, descomponer la base de datos monolítica compartida entre múltiples funcionalidades.

Diversas investigaciones han propuesto técnicas para la identificación de candidatos a microservicios mediante análisis estático, análisis dinámico, minería de repositorios de

software, clustering y análisis de dependencias. Sin embargo, muchas de estas propuestas se concentran principalmente en la descomposición de la lógica de aplicación, prestando menor atención al problema de la descomposición de la base de datos, el cual constituye uno de los principales factores de acoplamiento en sistemas monolíticos.

En este contexto, esta investigación de desarrollo tecnológico propone un método para la descomposición de bases de datos monolíticas hacia una arquitectura de microservicios bajo el enfoque de una base de datos por servicio. El método se fundamenta en el análisis semiautomático del código fuente mediante técnicas de análisis léxico-sintáctico y en la identificación de secuencias interactivas entre clases, métodos y elementos del modelo de datos que participan en la atención de una meta de valor o caso de uso. A partir de dichas secuencias se identifican agrupaciones funcionales que permiten proponer candidatos a microservicios y sus correspondientes bases de datos, reduciendo el acoplamiento existente y facilitando el proceso de modernización de aplicaciones empresariales.

Capítulo 2: Antecedentes

2.1 Planteamiento del problema

La autonomía de los servicios constituye uno de los principios fundamentales de la arquitectura de microservicios. Sin embargo, el aislamiento entre servicios se ve comprometido cuando éstos comparten una misma base de datos. La carencia de aislamiento conduce a carencia de autonomía entre microservicios debido a un acoplamiento indirecto entre éstos al seguir compartiendo el modelo de datos. De esta forma sólo se obtienen los llamados microlitos. Este acoplamiento conduce a fragilidad de las aplicaciones basadas en microservicios cuando hay extensiones y/o cambios a los requerimientos iniciales. La fragilidad se presenta cuando cambios en un microservicio pueden producir fallos en otros microservicios como efecto colateral.

Aunque existen diversos enfoques para identificar candidatos a microservicios, la descomposición de la base de datos continúa siendo una actividad compleja que depende en gran medida del conocimiento y experiencia de los arquitectos de software. Por ello, se requiere un método que permita identificar de manera sistemática las agrupaciones de funcionalidades y datos asociadas a una capacidad de negocio (caso de uso) en un sistema monolítico mediante el análisis de las secuencias de interacción entre clases, métodos y elementos del modelo de datos, facilitando la descomposición de la base de datos monolítica y contribuyendo a disminuir el acoplamiento indirecto entre los microservicios resultantes.

2.2 Justificación

La modernización de sistemas monolíticos constituye una necesidad estratégica para las organizaciones que buscan incrementar su capacidad de adaptación ante cambios tecnológicos y requerimientos de negocio cada vez más dinámicos. Aunque las arquitecturas de

microservicios se han consolidado como una alternativa ampliamente adoptada para el desarrollo de aplicaciones modernas, la transición desde sistemas monolíticos existentes continúa siendo un proceso complejo, costoso y altamente dependiente del conocimiento de expertos.

Uno de los principales desafíos en esta transición radica en la identificación de límites apropiados para los microservicios y en la descomposición de la base de datos monolítica compartida. En muchos sistemas empresariales, múltiples módulos funcionales acceden simultáneamente a las mismas entidades de datos, generando dependencias que dificultan la separación de responsabilidades y favorecen la aparición de acoplamiento indirecto entre componentes.

Las propuestas existentes suelen enfocarse en la identificación de microservicios a partir de dependencias estructurales o patrones arquitectónicos, sin abordar de manera integral la relación existente entre la lógica de negocio y los elementos de datos persistentes del sistema. Como consecuencia, las organizaciones enfrentan riesgos de inconsistencia, duplicación de información y degradación funcional durante los procesos de migración.

La presente investigación busca contribuir a la solución de este problema mediante el desarrollo de un método y su automatización, capaz de analizar el código fuente de sistemas monolíticos escritos en un lenguaje de programación e identificar las secuencias de interacción entre clases, métodos y elementos del modelo de datos involucrados en la ejecución de casos de uso específicos. Estas secuencias permiten determinar agrupaciones cohesivas de funcionalidades y datos, facilitando la definición de candidatos a microservicios y la asignación de una base de datos por servicio.

La relevancia de esta propuesta radica en que incorpora explícitamente la perspectiva de los datos dentro del proceso de descomposición arquitectónica, permitiendo reducir el acoplamiento derivado de bases de datos compartidas y proporcionando un mecanismo sistemático para apoyar la modernización de aplicaciones empresariales. Asimismo, la automatización del análisis mediante técnicas léxico-sintácticas contribuye a disminuir el esfuerzo manual requerido por arquitectos y desarrolladores, incrementando la reproducibilidad y objetividad del proceso de migración.

La ausencia de métodos sistemáticos que permitan analizar de manera conjunta las funcionalidades del sistema y sus elementos de datos, favorece la conservación de dependencias entre servicios, incrementando el riesgo de generar arquitecturas con bajo nivel de autonomía y alta dependencia entre los microservicios resultantes.

2.2.1 Situación Causal

La principal causa del acoplamiento indirecto entre microservicios es la permanencia de dependencias sobre una base de datos compartida durante los procesos de migración desde sistemas monolíticos. Esta situación se origina debido a la dificultad para identificar los límites

adecuados de servicio y determinar qué elementos del modelo de datos deben pertenecer a cada microservicio.

Adicionalmente, la complejidad de las relaciones existentes entre clases, métodos, funciones y entidades de datos dificulta la identificación de agrupaciones cohesivas que representen capacidades específicas de negocio. Como consecuencia, la definición de los límites de servicio y la asignación de los datos suelen depender del conocimiento y experiencia de los arquitectos de software.

2.3 Objetivos

2.3.1 Objetivo general

“Reducir el acoplamiento indirecto entre los servicios que integran aplicaciones de sistemas legados en operación, causado por compartir el mismo almacén de datos al momento de su migración hacia microservicios, mediante la aplicación de un método de descomposición de la base de datos”.

2.3.2 Objetivos específicos

- Alcanzar mayor grado de aislamiento de los servicios obtenidos de sistemas legados en operación mediante la aplicación del método de separación de su base de datos y comprobarlo a través del desarrollo de una métrica que mida el índice de acoplamiento entre los microservicios y sus tablas.
- Mantener la consistencia de datos que son compartidos por más de un microservicio, y que están replicados en cada una de las tablas de su base de datos, mediante patrones de comunicación como lo son: el patrón observador y/o patrón de eventos y mensajería.
- Aplicar el método propuesto en sistemas monolíticos reales mediante una herramienta informática que automatice el método de descomposición de la base de datos, con el propósito de determinar su aplicabilidad y verificar los resultados obtenidos en sistemas de diferente tamaño y complejidad.

2.4 Alcances y limitaciones

Alcances

- Desarrollo de un método para la descomposición de bases de datos de sistemas monolíticos legados o existentes, con el propósito de facilitar su transición hacia una arquitectura de microservicios bajo el enfoque de una base de datos por servicio.

- Identificación de agrupaciones de funcionalidades y elementos del modelo de datos mediante el análisis de las secuencias de interacción entre clases, métodos y entidades asociadas a capacidades de negocio.
- Evaluación del efecto del método propuesto mediante la comparación del índice de acoplamiento entre el sistema monolítico original y la arquitectura de microservicios obtenida tras la aplicación del método, utilizando una métrica que cuantifique la reducción del acoplamiento derivada del aislamiento de las bases de datos de cada microservicio.
- Análisis de los resultados obtenidos utilizando técnicas estadísticas que permitan determinar la efectividad del método propuesto.
- Implementación de una herramienta informática prototipo que apoye la ejecución automatizada del método, sujeta a la disponibilidad de tiempo y recursos durante el desarrollo de la investigación.
- Descomposición lógica de la base de datos monolítica, sin modificar la estructura interna de las entidades o atributos que conforman el esquema de datos original.

Limitaciones

- El método se aplica a sistemas monolíticos que estén desarrollados en capas, pero su implementación en la herramienta se limita a sistemas legados escritos en lenguaje Java.
- El código fuente objeto de análisis debe encontrarse en un estado funcional, libre de errores de compilación que impidan su procesamiento mediante las técnicas de análisis propuestas.
- El método no contempla actividades de depuración, corrección o refactorización del código fuente previo al proceso de análisis.
- La investigación no aborda la identificación ni la corrección de defectos de software existentes en los sistemas monolíticos analizados.
- El método se limita a la identificación de agrupaciones de funcionalidades y datos para apoyar la descomposición de la base de datos, sin considerar aspectos relacionados con la administración de la infraestructura de microservicios.
- No se incluyen actividades relacionadas con mecanismos de descubrimiento de servicios, balanceo de carga, seguridad, orquestación, contenerización o monitoreo propios de una arquitectura de microservicios.

- La investigación no contempla el diseño ni la implementación de mecanismos de comunicación entre los microservicios resultantes, sino que se limita a sugerir un patrón de comunicación entre los microservicios resultantes.
- La descomposición propuesta se encuentra orientada a sistemas que utilizan bases de datos relacionales y no considera bases de datos NoSQL u otros modelos de persistencia.
- El método se enfoca en la descomposición de la base de datos y la identificación de candidatos¹ a microservicios, por lo que no pretende automatizar completamente el proceso de migración de un sistema monolítico hacia una arquitectura de microservicios.
- En esta investigación, un sistema monolítico se define como una aplicación cuya funcionalidad se ejecuta como una única unidad desplegable y cuya persistencia se basa en una base de datos compartida. La tecnología empleada para implementar la capa de presentación (interfaces web, APIs REST, servicios SOAP o interfaces gráficas) no modifica esta clasificación arquitectónica, siempre que dichos mecanismos formen parte de la misma aplicación y compartan la misma lógica de negocio y la misma base de datos.

2.5 Fundamentos

2.5.1 Marco teórico

Ingeniería de Software y Modernización de Sistemas

La ingeniería de software es una disciplina que aplica principios científicos, metodologías y herramientas para el desarrollo, operación y mantenimiento de sistemas de software de manera sistemática y controlada. Su objetivo principal consiste en producir software de calidad que satisfaga los requerimientos funcionales y no funcionales de los usuarios, manteniendo criterios de confiabilidad, mantenibilidad y escalabilidad durante todo su ciclo de vida (Sommerville, 2016).

A medida que las organizaciones han incrementado su dependencia de los sistemas de información para soportar procesos de negocio críticos, la complejidad de las aplicaciones de software también ha aumentado considerablemente. Muchas de estas aplicaciones fueron desarrolladas utilizando arquitecturas monolíticas. Los sistemas desarrollados bajo estas condiciones suelen convertirse en sistemas legados, caracterizados por una alta dependencia de tecnologías específicas, documentación limitada, complejas relaciones entre componentes y una elevada dificultad para adaptarse a nuevos requerimientos del negocio. Debido a que estos sistemas continúan soportando procesos organizacionales fundamentales, su reemplazo

¹ En lo sucesivo de este documento, al usar la palabra “candidato(s)” de manera implícita, se referirá explícitamente a candidato(s) a microservicio(s).

completo resulta frecuentemente inviable desde una perspectiva técnica y económica (Sommerville, 2016).

Ante esta problemática surge la modernización de software, entendida como el conjunto de actividades orientadas a transformar sistemas existentes con el propósito de mejorar sus características técnicas y funcionales, preservando al mismo tiempo el conocimiento y las reglas de negocio acumuladas durante años de operación. La modernización puede involucrar diferentes estrategias, incluyendo la reestructuración del código fuente, la migración tecnológica, la reingeniería de procesos y la adopción de nuevas arquitecturas de software (Chikofsky & Cross, 1990).

Sistemas Monolíticos

La arquitectura monolítica ha sido durante décadas uno de los enfoques predominantes para el desarrollo de aplicaciones empresariales. En este tipo de arquitectura, las funcionalidades del sistema se implementan como una única unidad lógica y desplegable, donde los diferentes componentes de la aplicación, tales como la interfaz de usuario, la lógica de negocio y el acceso a datos, se encuentran integrados dentro de un mismo proceso (Fowler, 2002).

Este enfoque ofrece diversas ventajas durante las etapas iniciales del desarrollo de software. Entre ellas destacan la simplicidad de implementación, la facilidad de despliegue y la posibilidad de realizar pruebas integrales dentro de un entorno único. Además, la comunicación entre componentes ocurre mediante llamadas internas, lo que reduce la complejidad asociada a mecanismos de comunicación distribuidos (Sommerville, 2016).

Con el paso del tiempo, han surgido múltiples estrategias orientadas a modernizar sistemas monolíticos mediante procesos de refactorización, reingeniería y migración hacia arquitecturas más flexibles. Entre estas alternativas, la arquitectura de microservicios ha adquirido una relevancia significativa debido a su capacidad para promover la independencia funcional, la escalabilidad y la evolución autónoma de los servicios (Richardson, 2025). Sin embargo, la transición hacia este paradigma requiere comprender previamente la estructura interna del sistema monolítico y las relaciones existentes entre sus componentes y datos, aspecto que constituye uno de los principales retos abordados en la presente investigación.

La figura 1 muestra un ejemplo de una arquitectura monolítica y en la tabla 1 se muestran las principales características de una arquitectura monolítica.

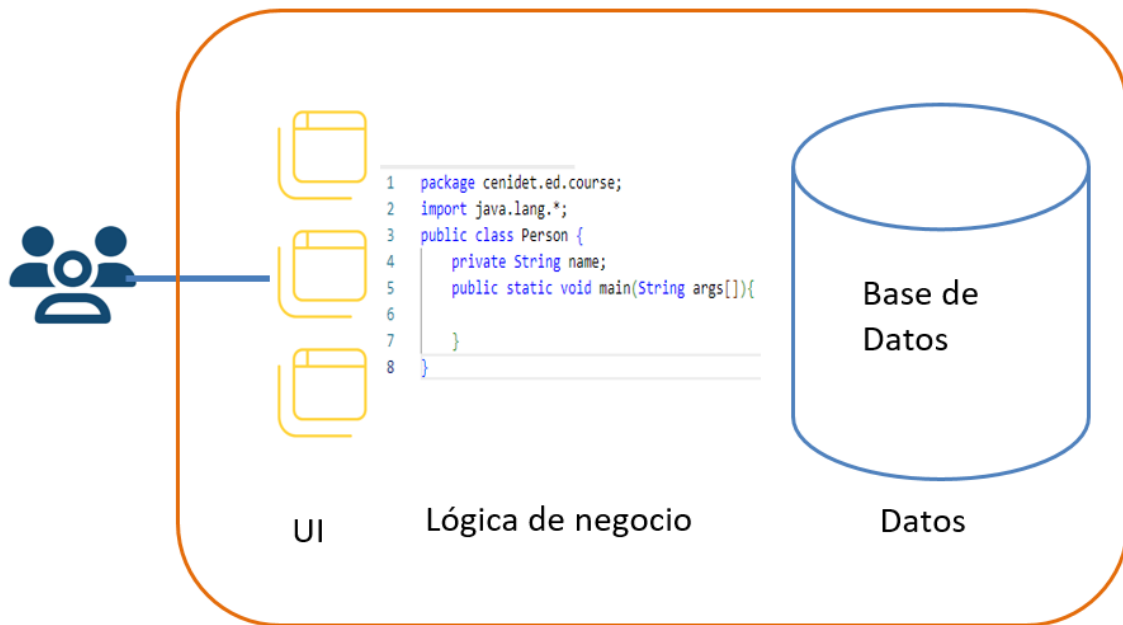


Figura 1 Ejemplo de la arquitectura monolítica usando el patrón de diseño MVC.

Tabla 1 Características de la Arquitectura Monolítica

Característica	Descripción
Despliegue	Único
Escalabilidad	Global
Acoplamiento	Medio a alto
Cohesión	Variable
Gestión de datos	Base de datos compartida
Mantenimiento	Complejo en sistemas grandes
Evolución tecnológica	Limitada

Arquitectura de Microservicios

La arquitectura de microservicios es un estilo que propone la construcción de aplicaciones como un conjunto de servicios pequeños, independientes y débilmente acoplados, donde cada servicio implementa una capacidad específica del negocio y puede desarrollarse, desplegarse y evolucionar de manera autónoma (Fowler & Lewis, s. f. 2014).

De acuerdo con (Newman, 2021), un microservicio puede definirse como un servicio autónomo que encapsula una funcionalidad de negocio claramente delimitada y que se comunica con otros servicios mediante mecanismos ligeros, generalmente basados en protocolos de red. La definición adecuada de los límites de servicio representa uno de los aspectos más relevantes dentro del diseño de arquitecturas de microservicios. En este contexto, los principios propuestos por el enfoque Domain-Driven Design (DDD) han adquirido una amplia aceptación como mecanismo para identificar fronteras alineadas con las capacidades del negocio (Evans, 2004).

Particularmente, el concepto de Bounded Context permite delimitar áreas funcionales que poseen modelos de dominio consistentes y relativamente independientes, favoreciendo la construcción de servicios altamente cohesivos (Evans, 2004), (Vernon, 2013).

No obstante, la migración hacia microservicios también introduce nuevos desafíos técnicos. La identificación de límites apropiados de servicio, la coordinación entre servicios distribuidos, la gestión de la consistencia de los datos y la definición de mecanismos de comunicación eficientes representan algunos de los problemas más estudiados en la literatura especializada (Dragoni et al., 2017), (Bass et al., 2021). Entre estos desafíos, la descomposición de la base de datos constituye una de las actividades más complejas debido a las múltiples dependencias existentes entre funcionalidades y elementos de información dentro de los sistemas monolíticos.

Diversos autores coinciden en que una arquitectura de microservicios exitosa debe buscar altos niveles de cohesión interna y bajos niveles de acoplamiento entre servicios (Newman, 2021), (Richardson, 2025), (Bass et al., 2021). Sin embargo, cuando varios microservicios comparten una misma base de datos, se generan dependencias que limitan su autonomía y dificultan la evolución independiente de los servicios. Por esta razón, la correcta identificación de los límites funcionales y de los datos asociados a cada capacidad de negocio se convierte en un factor crítico durante los procesos de migración desde sistemas monolíticos. En la figura 2 se muestra la arquitectura de microservicios, donde a cada servicio le corresponde una única base de datos.

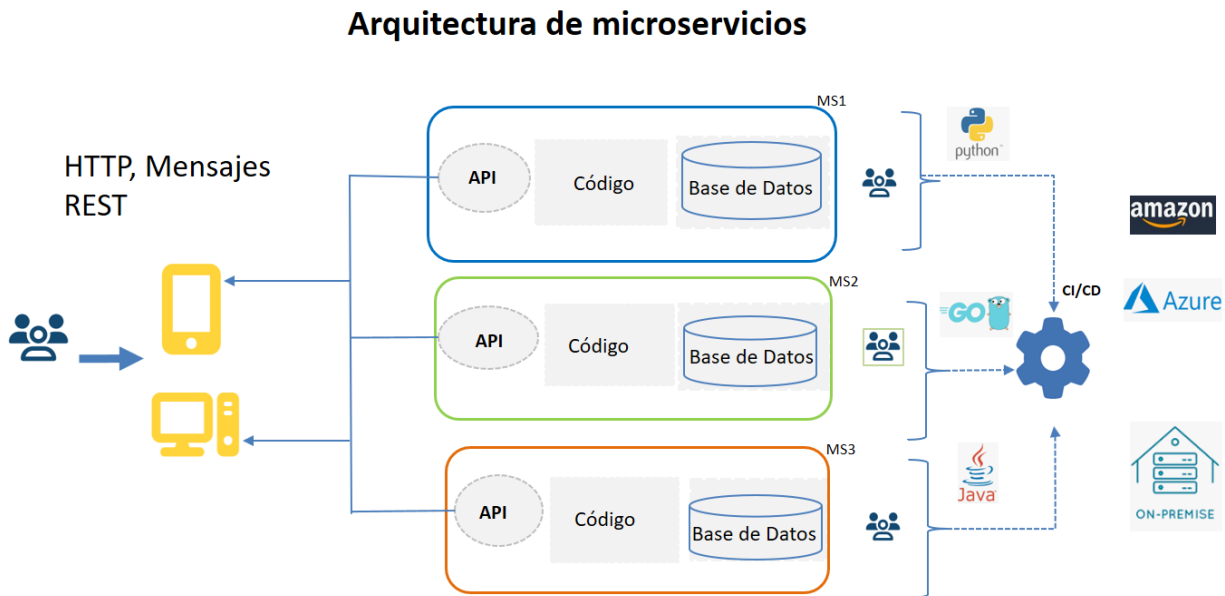


Figura 2 Principio de una base de datos por servicio en arquitecturas de microservicios.

La tabla 2 muestra las principales características de una arquitectura monolítica contrastada con la arquitectura de microservicios.

Tabla 2 Comparación entre arquitecturas monolíticas y arquitecturas de microservicios.

Característica	Monolito	Microservicios
Despliegue	Único	Independiente
Escalabilidad	Global	Por servicio
Gestión de datos	Compartida	Por servicio
Acoplamiento	Alto	Bajo
Cohesión	Variable	Alta
Evolución tecnológica	Limitada	Flexible

Acoplamiento y Cohesión

El acoplamiento y la cohesión constituyen dos de los principios fundamentales para el diseño y evaluación de sistemas de software. Ambos conceptos fueron introducidos formalmente por Stevens, Myers y Constantine (Stevens et al., 1974) como mecanismos para analizar la calidad estructural de los sistemas y continúan siendo utilizados como criterios esenciales para el diseño de arquitecturas modernas.

La cohesión se refiere al grado en que los elementos que conforman un componente de software contribuyen al cumplimiento de una misma responsabilidad o propósito funcional (Stevens et al., 1974). Un componente altamente cohesivo concentra funcionalidades estrechamente relacionadas y alineadas con un objetivo específico del negocio, favoreciendo la comprensión, mantenimiento y evolución del sistema. En arquitecturas de microservicios, la cohesión constituye una característica fundamental debido a que cada servicio debe representar una capacidad de negocio claramente delimitada y poseer responsabilidades bien definidas (Newman, 2021), (Evans, 2004).

Por otra parte, el acoplamiento representa el grado de dependencia existente entre componentes o módulos de software (Stevens et al., 1974). Un alto nivel de acoplamiento implica que los cambios realizados en un componente pueden afectar el comportamiento de otros componentes dependientes, incrementando la complejidad del mantenimiento y reduciendo la flexibilidad del sistema. En consecuencia, uno de los objetivos fundamentales del diseño arquitectónico consiste en minimizar las dependencias entre componentes mientras se maximiza la cohesión interna de cada unidad funcional (Bass et al., 2021).

Dentro del contexto de las arquitecturas de microservicios, el bajo acoplamiento adquiere una relevancia particular debido a la necesidad de garantizar la autonomía de los servicios. Según (Newman, 2021), la independencia de despliegue, mantenimiento y evolución constituye una de las principales ventajas de este paradigma arquitectónico. Sin embargo, dicha independencia puede verse comprometida cuando los servicios mantienen dependencias excesivas sobre recursos compartidos, interfaces inestables o modelos de datos comunes.

Particularmente, cuando las dependencias entre servicios se producen a través de elementos compartidos del modelo de datos, se presenta un fenómeno conocido como acoplamiento indirecto. Este tipo de acoplamiento ocurre cuando dos o más servicios no interactúan explícitamente entre sí, pero dependen de los mismos elementos de información para ejecutar

sus funcionalidades. Como resultado, modificaciones realizadas sobre una entidad, tabla o estructura de datos pueden generar efectos colaterales en servicios aparentemente independientes.

El acoplamiento indirecto representa uno de los principales obstáculos para alcanzar la autonomía deseada en una arquitectura de microservicios. La presencia de dependencias implícitas dificulta la evolución independiente de los servicios, incrementa los costos de mantenimiento y aumenta el riesgo de introducir errores durante la implementación de cambios. En casos extremos, esta situación conduce a la formación de sistemas conocidos como microlitos (Boner, 2019.), en los cuales la aplicación aparenta estar dividida en microservicios, pero conserva características de acoplamiento propias de una arquitectura monolítica (Richardson, s. f.). En la tabla 3 se visualizan las relaciones que existen entre la cohesión, el acoplamiento y la autonomía de los servicios.

Tabla 3 Relación entre cohesión, acoplamiento y autonomía de servicios.

Concepto	Descripción	Impacto en Microservicios
Cohesión	Relación funcional entre elementos de un componente	Favorece servicios especializados
Acoplamiento	Dependencia entre componentes	Reduce autonomía
Acoplamiento de datos	Dependencia mediante información compartida	Afecta evolución independiente
Acoplamiento indirecto	Dependencia a través de una base de datos común	Genera microlitos

Base de Datos Compartida y Base de Datos por Servicio

La gestión de los datos constituye uno de los aspectos críticos dentro del diseño de arquitecturas de software. En los sistemas monolíticos tradicionales, las funcionalidades de la aplicación suelen compartir una única base de datos que centraliza toda la información requerida por los diferentes módulos del sistema. Este enfoque facilita la implementación de transacciones, garantiza la integridad referencial y simplifica el acceso a los datos durante las primeras etapas de desarrollo (Fowler, 2002).

Cuando las organizaciones emprenden procesos de migración hacia arquitecturas de microservicios, la existencia de una base de datos globalmente compartida se convierte en uno de los principales obstáculos para alcanzar la autonomía de los servicios. Aunque la lógica de negocio pueda dividirse en múltiples servicios independientes, la permanencia de una única base de datos mantiene dependencias implícitas entre ellos, limitando los beneficios esperados de la arquitectura distribuida (Richardson, 2025).

La arquitectura de microservicios promueve el principio de propiedad de los datos (Data Ownership), según el cual cada servicio es responsable de la gestión y persistencia de la información necesaria para implementar su capacidad de negocio (Richardson, 2025). Este principio establece que ningún servicio debe acceder directamente a la base de datos de otro servicio, sino que cualquier intercambio de información debe realizarse mediante interfaces o mecanismos de comunicación explícitamente definidos.

Con base en este principio surge el patrón denominado Base de Datos por Servicio (Database per Service), ampliamente reconocido dentro de la literatura especializada sobre microservicios (Richardson, 2025). Este patrón propone que cada microservicio disponga de su propio mecanismo de persistencia y mantenga control exclusivo sobre los datos asociados a su dominio funcional. Como resultado, los servicios pueden evolucionar de manera independiente, modificar sus estructuras de almacenamiento y adoptar tecnologías específicas sin afectar a otros componentes del sistema.

No obstante, la adopción de una base de datos por servicio también introduce nuevos desafíos. La descomposición de los datos obliga a replantear aspectos relacionados con la consistencia de la información, las transacciones distribuidas y la sincronización entre servicios (Helland, 2007). Problemas que anteriormente eran resueltos mediante mecanismos transaccionales centralizados deben abordarse ahora mediante estrategias específicas para sistemas distribuidos, tales como eventos de dominio, consistencia eventual y patrones de coordinación de servicios (Richardson, 2025).

Uno de los mayores retos durante la transición desde sistemas monolíticos hacia arquitecturas de microservicios consiste precisamente en determinar qué elementos de información deben pertenecer a cada servicio. En aplicaciones empresariales complejas, las entidades de datos suelen participar en múltiples procesos de negocio y mantener relaciones con numerosas funcionalidades del sistema. Esta situación dificulta la identificación de límites adecuados para la separación de los datos y aumenta el riesgo de conservar dependencias no deseadas entre los servicios resultantes (Klock et al., 2019), (Mazlami et al., 2017).

Tabla 4 Comparación entre los enfoques de base de datos compartida y base de datos por servicio.

Característica	Base de Datos Compartida	Base de Datos por Servicio
Propiedad de datos	Compartida	Exclusiva
Acoplamiento	Alto	Bajo
Autonomía de servicios	Limitada	Alta
Evolución independiente	Difícil	Facilitada
Escalabilidad	Global	Por servicio
Consistencia	Centralizada	Distribuida

Análisis Estático de Código

El análisis estático de código constituye una técnica ampliamente utilizada en ingeniería de software para examinar programas sin necesidad de ejecutarlos. Su objetivo consiste en obtener información relacionada con la estructura, comportamiento potencial, dependencias y características internas de un sistema mediante la inspección directa de su código fuente (Aho et al., 2006), (Cooper & Torczon, 2022).

A diferencia del análisis dinámico, que requiere la ejecución del software para recopilar información sobre su comportamiento, el análisis estático permite estudiar aplicaciones completas a partir de los artefactos disponibles durante el desarrollo, incluyendo archivos

fuentes, modelos, configuraciones y estructuras de datos. Esta característica resulta particularmente valiosa en sistemas legados, donde frecuentemente existe documentación limitada o desactualizada (Ducasse & Pollet, 2009).

Dentro de los procesos de modernización de software, el análisis estático desempeña un papel fundamental debido a que permite recuperar conocimiento implícito contenido en el código fuente. Diversos autores señalan que gran parte de la información relacionada con la arquitectura, dependencias funcionales y reglas de negocio de un sistema se encuentra distribuida entre clases, métodos, estructuras de datos y componentes de software (Ducasse & Pollet, 2009). Como consecuencia, el análisis estático se ha convertido en una de las principales técnicas utilizadas para apoyar actividades de recuperación de arquitectura y reingeniería.

Uno de los principales beneficios de este enfoque consiste en la posibilidad de identificar relaciones estructurales entre los distintos elementos del sistema. A través del análisis del código fuente es posible detectar dependencias entre clases, llamadas a métodos, relaciones de herencia, asociaciones, composición y acceso a recursos de persistencia. Esta información proporciona una representación detallada de la organización interna del software y puede emplearse para identificar agrupaciones funcionales potencialmente asociadas a capacidades de negocio (Koschke, 2005). En la tabla 5 se realiza una comparación entre las técnicas estáticas y dinámicas en ingeniería de software.

Tabla 5 Comparación entre técnicas de análisis de software.

Técnica	Requiere Ejecución	Información Obtenida
Análisis Estático	No	Estructura y dependencias
Análisis Dinámico	Sí	Comportamiento en ejecución
Híbrido	Parcial	Estructura y comportamiento

Análisis Léxico y Sintáctico

El análisis léxico y sintáctico constituye una de las etapas fundamentales dentro del proceso de compilación y procesamiento automático de lenguajes de programación. Su objetivo consiste en transformar el código fuente escrito por un desarrollador en estructuras formales que puedan ser interpretadas y analizadas por herramientas de software (Aho et al., 2006).

Los lenguajes de programación suelen describirse mediante gramáticas formales que especifican las reglas sintácticas que deben cumplir los programas válidos. Estas gramáticas permiten definir de manera precisa la estructura de los distintos elementos del lenguaje, tales como clases, métodos, expresiones, sentencias y declaraciones (Aho et al., 2006), (Cooper & Torczon, 2022).

El análisis léxico representa la primera fase de procesamiento de un programa. Durante esta etapa, el código fuente es examinado carácter por carácter para identificar unidades mínimas denominadas tokens. Los tokens representan elementos significativos del lenguaje, tales como identificadores, palabras reservadas, operadores, literales y símbolos especiales (Aho et al., 2006).

Por ejemplo, la instrucción:

```
Cliente cliente = new Cliente();
```

puede descomponerse en una secuencia de tokens que representan identificadores, operadores de asignación, palabras reservadas y delimitadores sintácticos. Esta transformación simplifica las etapas posteriores del análisis al convertir el flujo de caracteres en una secuencia estructurada de elementos reconocibles por el compilador.

Una vez obtenida la secuencia de tokens, se realiza el análisis sintáctico. Esta fase verifica que la estructura del programa cumpla con las reglas definidas por la gramática del lenguaje. El resultado de este proceso es una representación jerárquica denominada árbol sintáctico (Parser Tree), donde cada nodo representa una construcción gramatical reconocida durante el análisis (Aho et al., 2006).

El árbol sintáctico permite representar explícitamente las relaciones existentes entre los diferentes elementos del programa. A partir de esta representación es posible identificar estructuras tales como clases, métodos, atributos, sentencias de control, invocaciones de métodos y expresiones complejas. Debido a estas características, los árboles sintácticos constituyen una fuente valiosa de información para actividades de análisis y comprensión de software.

Para algunos procesos de análisis resulta conveniente utilizar una representación más compacta y orientada a la semántica del programa. Con este propósito se emplean los árboles de sintaxis abstracta (Abstract Syntax Trees, AST), los cuales eliminan detalles sintácticos innecesarios y conservan únicamente los elementos relevantes para el análisis (Parr, 2013). Esta simplificación facilita la extracción de información estructural y reduce la complejidad de los algoritmos de procesamiento.

Dentro del contexto de la modernización de software, el análisis léxico y sintáctico permite automatizar la recuperación de conocimiento contenido en aplicaciones legadas. Esta capacidad resulta particularmente importante cuando la documentación del sistema es insuficiente o se encuentra desactualizada, situación frecuente en aplicaciones empresariales desarrolladas y mantenidas durante largos periodos de tiempo (Ducasse & Pollet, 2009).

En este trabajo de investigación, las técnicas de análisis léxico y sintáctico son utilizadas para construir representaciones estructuradas del código fuente a partir de las cuales se identifican clases, métodos y elementos relacionados con el modelo de datos. Estas representaciones constituyen la base para la obtención de las secuencias de interacción de casos de uso utilizadas posteriormente en el proceso de identificación de candidatos a microservicios y descomposición de bases de datos. En la tabla 6 se presentan las etapas del procesamiento de lenguajes de programación.

Tabla 6 Etapas fundamentales del procesamiento de lenguajes de programación.

Etapa	Entrada	Salida
Análisis Léxico	Código fuente	Tokens
Análisis Sintáctico	Tokens	Parse Tree
Construcción de AST	Parse Tree	AST
Análisis Semántico	AST	Información estructural

ANTLR y Patrón Visitor

La extracción automática de información contenida en el código fuente requiere mecanismos capaces de interpretar la estructura sintáctica de los programas y recorrer sistemáticamente los elementos que los conforman. Para este propósito, diversas herramientas de procesamiento de lenguajes utilizan gramáticas formales y estructuras de análisis que permiten transformar el código fuente en representaciones internas susceptibles de ser analizadas automáticamente (Parr, 2013), (Aho et al., 2006).

Una de las herramientas más utilizadas para la construcción de analizadores léxicos y sintácticos es ANTLR (*ANother Tool for Language Recognition*). ANTLR es un generador de analizadores que permite definir gramáticas formales para distintos lenguajes de programación y producir automáticamente los componentes necesarios para realizar análisis léxico, sintáctico y semántico de programas (Parr, 2013).

El funcionamiento de ANTLR se basa en la definición de una gramática que describe las reglas sintácticas del lenguaje objetivo. A partir de dicha gramática, la herramienta genera automáticamente un analizador léxico (Lexer) y un analizador sintáctico (Parser), los cuales permiten reconocer la estructura del código fuente y construir árboles sintácticos que representan la organización lógica del programa (Parr, 2013).

En el caso del lenguaje Java, existen gramáticas ampliamente utilizadas que permiten identificar elementos como paquetes, clases, interfaces, atributos, métodos, anotaciones e instrucciones de control. Estas capacidades convierten a ANTLR en una herramienta adecuada para actividades de análisis estático, recuperación de arquitectura e ingeniería inversa de software (Parr, 2013).

Por otro lado, el patrón Visitor fue propuesto por Gamma et al. (Gamma et al., 1994) como uno de los patrones de diseño orientados al comportamiento. Su propósito consiste en separar los algoritmos que operan sobre una estructura de objetos de las clases que conforman dicha estructura, permitiendo agregar nuevas funcionalidades sin modificar los elementos originales del sistema.

Aplicado al contexto del análisis sintáctico, el patrón Visitor permite recorrer cada nodo del árbol generado por ANTLR e implementar acciones específicas para cada tipo de construcción gramatical identificada. Por ejemplo, es posible definir métodos especializados para procesar declaraciones de clases, invocaciones de métodos, atributos o instrucciones de acceso a datos (Parr, 2013).

Esta característica resulta especialmente útil en actividades de análisis de código fuente, ya que permite construir mecanismos flexibles para recuperar información estructural y funcional del sistema. Mediante la implementación de visitantes especializados es posible identificar relaciones entre clases, dependencias entre métodos y accesos a entidades del modelo de datos, generando representaciones que facilitan la comprensión de sistemas complejos.

En el contexto de la presente investigación, ANTLR se utiliza para construir representaciones sintácticas de aplicaciones monolíticas desarrolladas en Java. Posteriormente, mediante la implementación del patrón Visitor, se recorren los árboles generados para identificar las clases, métodos y elementos del modelo de datos que participan en la atención de funcionalidades específicas del sistema. En la tabla 7 se observa la interacción entre ANTLR y el patrón Visitor en una secuencia de un caso de uso.

Tabla 7 Uso de ANTLR y el patrón Visitor para la obtención de secuencias de interacción.

Componente	Función
Gramática	Define las reglas del lenguaje
Lexer	Genera tokens
Parser	Construye el árbol sintáctico
Parse Tree	Representa la estructura gramatical
AST	Representa la estructura lógica del programa
Visitor	Recorre el árbol y extrae información
Secuencia de Interacción	Representa colaboraciones entre elementos del sistema

Descomposición de Bases de Datos

La descomposición de una base de datos monolítica implica descomponer o dividir una base de datos centralizada en múltiples bases de datos más pequeñas, cada una asociada a un microservicio específico. Este proceso debe alinearse con el principio de diseño orientado al dominio (DDD, por sus siglas en inglés), que enfatiza la identificación de contextos acotados (bounded contexts) para delimitar las responsabilidades de cada servicio (Evans, 2004). Según (Richardson, 2025), la descomposición de bases de datos debe considerar los siguientes aspectos:

- **Identificación de dominios:** Utilizar técnicas como Domain Driven Design para identificar subdominios dentro de la aplicación que pueden mapearse a microservicios.
- **Separación lógica:** Dividir las tablas de la base de datos monolítica en grupos que correspondan a los dominios identificados, minimizando el acoplamiento entre ellos.
- **Gestión de consistencia:** Adoptar estrategias como la consistencia eventual o el uso de patrones como Saga o Eventos para manejar transacciones distribuidas.
- **Optimización de rendimiento:** Seleccionar el tipo de base de datos (relacional, NoSQL, etc.) que mejor se adapte a las necesidades de cada microservicio.

El proceso de descomposición debe ser gradual para evitar interrupciones en el sistema. Todkar (2020) propone un enfoque iterativo que incluye la separación de un servicio del monolito, la creación de tablas específicas para ese servicio, y la migración progresiva de los datos hacia una nueva base de datos. Este enfoque reduce el riesgo de pérdida de datos y asegura la continuidad del servicio durante la migración.

Métodos y Herramientas para la Descomposición

La literatura reciente destaca varios métodos y herramientas para facilitar la descomposición de bases de datos. Por ejemplo, Prado Valero et al. (2021) desarrollaron una herramienta de división que utiliza grafos para representar las dependencias entre componentes del sistema monolítico y generar microservicios automáticamente. Esta herramienta emplea análisis de contexto, diseño estructurado, y pruebas para verificar si la partición es correcta. De manera similar, (Vera-Rivera et al., 2020) proponen el uso de algoritmos genéticos y aprendizaje automático no supervisado para determinar la granularidad óptima de los microservicios, lo que incluye la asignación de datos a cada servicio.

Otros enfoques, como el propuesto por (Gysel et al., 2016) , utilizan métricas de acoplamiento y cohesión para evaluar la calidad de la descomposición de sistemas. Estas métricas permiten identificar particiones que minimizan las dependencias entre servicios y maximizan la autonomía. Además, herramientas como Spring Boot y frameworks de orquestación como Kubernetes facilitan la implementación de bases de datos distribuidas en entornos de microservicios (Valero et al., 2021).

Desafíos en la Migración

La migración de una base de datos monolítica a microservicios enfrenta varios desafíos, incluyendo:

- **Integridad referencial:** En un sistema monolítico, las claves foráneas aseguran la integridad de los datos. En microservicios, la falta de claves foráneas requiere estrategias alternativas, como la validación de la lógica del servicio o el uso de eventos para mantener la consistencia (Morling, 2023).
- **Complejidad operativa:** La gestión de múltiples bases de datos aumenta la complejidad en términos de monitoreo, respaldo, y recuperación ante fallos.
- **Consistencia de datos:** La transición del modelo ACID al modelo BASE (Basically Available, Soft state, Eventual consistency) requiere un cambio en la forma en que se diseñan las transacciones (Kleppmann, 2017).
- **Resistencia al cambio:** Los equipos de desarrollo pueden enfrentar dificultades para adaptarse a los nuevos paradigmas de desarrollo y operación, lo que subraya la importancia de una cultura DevOps.

Patrones de Diseño Relevantes

Para abordar estos desafíos, se han propuesto varios patrones de diseño en la literatura:

- **Patrón Database per Service:** Cada microservicio tiene su propia base de datos, lo que promueve la autonomía y reduce el acoplamiento (Fowler, 2018).
- **Patrón Saga:** Utilizado para gestionar transacciones distribuidas, dividiendo una transacción en pasos locales que pueden compensarse en caso de fallo (Richardson, 2025).
- **Patrón Event Sourcing:** Los cambios en los datos se almacenan como una secuencia de eventos de comunicación, lo que facilita la consistencia eventual y la auditoría (Fowler, 2018).
- **Patrón CQRS (Command Query Responsibility Segregation):** Separa las operaciones de lectura y escritura, permitiendo optimizar las bases de datos para cada tipo de operación (Young, 2010).

2.6 Estado del arte

En este capítulo se presenta una revisión de las investigaciones y propuestas relevantes relacionadas con la migración de sistemas monolíticos hacia arquitecturas de microservicios. Se enfatiza en los métodos de descomposición de bases de datos e identificación de candidatos a microservicios. El análisis de estos trabajos permite identificar las técnicas, enfoques y herramientas empleadas por la comunidad científica para abordar los problemas de acoplamiento, autonomía de servicios y separación de la base de datos durante los procesos de modernización de software.

Asimismo, esta revisión permite reconocer las fortalezas y limitaciones de las propuestas existentes, así como las áreas que aún requieren investigación. En particular, se busca identificar el grado en que los trabajos actuales consideran las relaciones entre funcionalidades, clases, métodos y elementos del modelo de datos para apoyar la descomposición de sistemas monolíticos.

2.6.1 Trabajos relacionados

Con el propósito de analizar de manera estructurada las investigaciones relacionadas con la presente tesis, los trabajos identificados en la literatura se agruparon en núcleos temáticos de acuerdo con el problema que abordan y su contribución al proceso de migración de sistemas monolíticos hacia arquitecturas de microservicios incluida la base de datos.

La clasificación propuesta permite identificar las principales líneas de investigación desarrolladas en el área para resolver problemas relacionados con la consistencia de datos, la

persistencia de información, la migración de sistemas legados, la descomposición de bases de datos y la identificación de candidatos a microservicios.

Asimismo, esta organización facilita el análisis comparativo de las propuestas existentes y permite identificar las limitaciones y oportunidades de investigación que justifican el desarrollo del método propuesto en esta tesis.

Consistencia, Aislamiento e Integridad de Datos en Arquitecturas de Microservicios

La consistencia e integridad de los datos representan algunos de los desafíos asociados a la adopción de arquitecturas de microservicios. A diferencia de los sistemas monolíticos, donde las transacciones suelen ejecutarse sobre una única base de datos centralizada, las arquitecturas de microservicios promueven la independencia de los servicios mediante el principio de base de datos por servicio. Esta separación incrementa la autonomía de los servicios, pero introduce dificultades relacionadas con la sincronización, consistencia e integridad de la información distribuida.

Diversos estudios han abordado esta problemática desde diferentes perspectivas. (Dragoni et al., 2017a) identifican la gestión de datos distribuidos como uno de los principales retos de las arquitecturas de microservicios, señalando que la autonomía de los servicios implica renunciar a mecanismos tradicionales de consistencia fuerte utilizados en sistemas monolíticos. Los autores destacan que la comunicación distribuida y la independencia de los modelos de datos generan nuevos desafíos para garantizar la integridad de la información.

En este mismo contexto, (Gaffney et al., 2021) proponen un mecanismo denominado “Database Isolation by Scheduling”, cuyo objetivo consiste en mejorar el aislamiento de las operaciones ejecutadas sobre bases de datos distribuidas. Los autores argumentan que la consistencia de los datos puede preservarse mediante estrategias de planificación que reduzcan los conflictos entre transacciones concurrentes. Los resultados obtenidos muestran mejoras en el nivel de aislamiento sin comprometer significativamente el rendimiento del sistema. Sin embargo, la propuesta se enfoca en la administración de transacciones una vez que los límites de servicio han sido definidos, sin abordar el problema de la identificación de dichos límites.

Por otra parte, (Zuckmantel et al., 2022) presentan un enfoque basado en eventos denominado “Event-Based Data-Centric Semantics for Consistent Data Management in Microservices”. La propuesta utiliza mecanismos de comunicación orientados a eventos para coordinar cambios de estado entre servicios distribuidos y mantener la consistencia de los datos. Los autores proponen diseñar un servicio de coordinación que se basa en una semántica formal centrada en datos y basada en eventos para microservicios donde se especifica la interacción de transacciones entre estos y sus respectivas bases de datos. Además, proporcionan una formalización de las propiedades de consistencia entre datos y describen cómo se pueden usar para respaldar el monitoreo dinámico de la información, proporcionando así sistemas de microservicios sólidos.

Adicionalmente, la literatura reporta el uso de patrones arquitectónicos como Saga, Event Sourcing y Command Query Responsibility Segregation (CQRS) para gestionar transacciones

distribuidas y mantener la consistencia eventual de los datos. Estos enfoques buscan coordinar operaciones entre múltiples servicios sin recurrir a mecanismos tradicionales de bloqueo o transacciones distribuidas, favoreciendo la escalabilidad y autonomía de los microservicios. Sin embargo, su aplicación se encuentra orientada principalmente a sistemas que ya han sido descompuestos en servicios independientes.

Las investigaciones revisadas se concentran principalmente en resolver problemas asociados a la gestión de datos distribuidos una vez que la arquitectura de microservicios ha sido definida. En contraste, existe una atención limitada al problema previo de determinar cómo deben identificarse los límites de los servicios y cómo debe realizarse la descomposición de una base de datos monolítica para minimizar dependencias futuras entre los servicios resultantes.

Por consiguiente, aunque estos trabajos proporcionan mecanismos relevantes para mantener la consistencia e integridad de la información en entornos distribuidos, no ofrecen criterios explícitos para identificar agrupaciones de funcionalidades y elementos de datos durante los procesos de migración desde sistemas monolíticos. Esta situación pone en evidencia la necesidad de métodos que permitan apoyar la definición de límites de servicio desde etapas tempranas del proceso de descomposición, contribuyendo a reducir el acoplamiento indirecto y facilitar la autonomía de los microservicios resultantes.

Persistencia y Base de Datos por Servicio

La gestión de la persistencia constituye uno de los aspectos fundamentales en las arquitecturas de microservicios debido a su estrecha relación con la autonomía, escalabilidad y evolución independiente de los servicios.

Diversos autores coinciden en que la separación de la persistencia representa un requisito indispensable para alcanzar un adecuado nivel de aislamiento entre servicios. (Dragoni et al., 2017) señalan que compartir una misma base de datos genera dependencias ocultas que limitan la autonomía de los servicios y dificultan su evolución independiente. Desde esta perspectiva, la persistencia deja de ser un recurso común para convertirse en una responsabilidad exclusiva de cada servicio.

En este contexto, (Munonye & Martinek, 2020) realizan una evaluación comparativa de diferentes patrones de almacenamiento utilizados en arquitecturas de microservicios, entre los que destacan Database per Service, Command Query Responsibility Segregation (CQRS) y Event Sourcing. Los resultados de su estudio muestran que el patrón Database per Service favorece el desacoplamiento entre servicios y mejora la capacidad de escalamiento independiente, mientras que CQRS y *Event Sourcing* proporcionan mecanismos adicionales para gestionar la consistencia y sincronización de información en entornos distribuidos. Sin embargo, los autores asumen que la descomposición de los servicios y la asignación de los datos a cada uno de ellos han sido previamente definidas, por lo que no abordan el problema de cómo identificar dichas particiones a partir de sistemas monolíticos existentes.

Por otra parte, (Kazanavičius et al., 2022) estudian la migración de bases de datos monolíticas hacia esquemas de persistencia distribuidos basados en arquitecturas de microservicios. Su propuesta plantea utilizar modelos de persistencia heterogéneos (Polyglot Persistence), donde cada servicio puede seleccionar la tecnología de almacenamiento más adecuada para sus necesidades funcionales y de rendimiento. Los autores concluyen que esta estrategia incrementa la flexibilidad y la capacidad de adaptación de los sistemas, aunque reconocen que la correcta identificación de los límites de datos continúa siendo uno de los principales desafíos durante la migración.

Asimismo, diversos trabajos destacan que la adopción del *principio de una base de datos por servicio* no resuelve por sí misma los problemas relacionados con la fragmentación de la información. La definición incorrecta de los límites de datos puede generar redundancia, inconsistencias y dependencias cruzadas entre servicios, reproduciendo problemas similares a los observados en bases de datos compartidas. En consecuencia, la efectividad de este principio depende en gran medida de la correcta identificación de las capacidades de negocio y de los datos asociados a cada servicio.

Por lo anterior, aunque el *principio de una base de datos por servicio* es ampliamente aceptado como una práctica recomendada en arquitecturas de microservicios, aún existe la necesidad de métodos que permitan identificar y fragmentar de forma adecuada los modelos de datos asociados a cada capacidad de negocio. Esta problemática resulta relevante en sistemas legados, donde las dependencias entre clases, métodos y entidades de datos suelen encontrarse fuertemente acopladas dentro de una única base de datos monolítica.

Migración de Sistemas Monolíticos hacia Arquitecturas de Microservicios

La migración de sistemas monolíticos hacia arquitecturas de microservicios constituye una de las principales estrategias de modernización adoptadas por las organizaciones para mejorar la escalabilidad, mantenibilidad y capacidad de evolución de sus sistemas de información.

Las primeras investigaciones en este ámbito se enfocan principalmente en la identificación de candidatos a microservicios mediante el análisis estructural del código fuente. Trabajos como los de (Levcovitz et al., 2016) y (Mazlami et al., 2017) proponen mecanismos para identificar agrupaciones funcionales a partir de dependencias entre componentes, clases y módulos. Estos enfoques demuestran que es posible obtener particiones coherentes del sistema mediante técnicas de análisis estático; sin embargo, su atención se concentra en la lógica de negocio y no considera explícitamente las dependencias asociadas al modelo de datos. Como consecuencia, los límites de servicio identificados pueden no corresponder con los límites de persistencia de datos requeridos para garantizar la autonomía de los microservicios.

Las investigaciones han evolucionado hacia enfoques que incorporan información adicional proveniente de la ejecución del sistema. En esta línea, (Krause et al., 2020), (Andrade et al., 2023) y (Matias et al., 2020) combinan análisis estático y dinámico con el propósito de obtener una visión más precisa de las interacciones entre componentes. Los resultados muestran mejoras en la identificación de límites funcionales al considerar tanto las dependencias estructurales

como el comportamiento observado durante la ejecución. No obstante, aunque estos enfoques permiten comprender mejor las relaciones existentes entre los componentes del sistema, continúan concentrándose en la delimitación de servicios y prestan escasa atención a la descomposición de la base de datos asociada a dichos servicios.

Otros trabajos han intentado reducir la intervención humana mediante mecanismos semiautomáticos. (A. Selmadji et al., 2020) proponen un enfoque que utiliza información derivada del código fuente para identificar límites potenciales de servicio. Aunque estas propuestas contribuyen a disminuir el esfuerzo requerido durante la transición arquitectónica, continúan dependiendo de la experiencia de los arquitectos para definir los límites finales de los servicios y de los datos asociados a cada uno de ellos.

La relevancia de este problema es confirmada por estudios empíricos y de mapeo sistemático. (Fritzsche et al., 2024) y (Taibi et al., 2020) reportan que la definición de límites de servicio y la separación de datos constituyen algunos de los principales desafíos enfrentados por las organizaciones durante los procesos de migración. De forma similar, (Martínez et al., 2025), mediante un estudio de mapeo sistemático, concluyen que la mayoría de las investigaciones se concentra en la identificación de microservicios a partir del análisis estructural, funcional o de dominio, mientras que la migración de la base de datos continúa siendo una problemática insuficientemente atendida por la literatura.

A partir del análisis de los trabajos revisados se identifican tres tendencias predominantes. La primera se orienta a la identificación de microservicios mediante análisis estructural del código fuente; la segunda incorpora información dinámica obtenida durante la ejecución del sistema; y la tercera utiliza conocimiento del dominio para definir límites de servicio. Aunque estos enfoques han demostrado ser efectivos para apoyar la descomposición funcional de sistemas monolíticos, ninguno integra de manera explícita el análisis simultáneo de capacidades de negocio, interacciones entre componentes y entidades de datos con el objetivo de obtener, de forma coordinada, tanto los candidatos a microservicios como los fragmentos de base de datos asociados.

En conclusión, la literatura deja en evidencia avances significativos en la migración de sistemas monolíticos hacia arquitecturas de microservicios; sin embargo, aún existe una brecha relacionada con la ausencia de métodos que permitan vincular de manera sistemática los elementos funcionales del sistema con los modelos de datos que soportan cada capacidad de negocio.

Descomposición de Bases de Datos en la Migración hacia Microservicios

La descomposición de la base de datos constituye uno de los desafíos más complejos durante la migración de sistemas monolíticos hacia arquitecturas de microservicios. Esta situación se debe a que las bases de datos monolíticas suelen concentrar información perteneciente a múltiples procesos de negocio, generando dependencias estructurales y semánticas que dificultan la definición de límites claros del modelo de datos.

La literatura coincide en que el principio de una base de datos por servicio constituye un mecanismo fundamental para favorecer la autonomía de los microservicios. Sin embargo, existe una diferencia entre reconocer la necesidad de separar los datos y disponer de mecanismos sistemáticos que permitan determinar cómo realizar dicha separación a partir de los límites funcionales del sistema. Diversos trabajos han estudiado estrategias para identificar límites de servicio o para fragmentar, distribuir y administrar los datos asociados a dichos límites. No obstante, estos procesos suelen abordarse de manera separada, por lo que persiste el desafío de establecer un procedimiento sistemático que vincule la identificación de los límites de servicio con la correspondiente descomposición de la base de datos de un sistema monolítico.

Algunos trabajos relacionados con la descomposición de datos abordan el problema desde una perspectiva estructural. Yue (2019) analiza diferentes estrategias de descomposición de información y destaca que la correcta partición de los datos constituye un factor determinante para preservar la coherencia de los procesos de negocio. El autor identifica que las dependencias entre entidades suelen extenderse más allá de las fronteras funcionales inicialmente definidas, lo que incrementa la complejidad de cualquier proceso de fragmentación. No obstante, el estudio se desarrolla desde una perspectiva general de descomposición de datos y no considera específicamente los desafíos asociados a la migración hacia microservicios.

Posteriormente, (Bandara & Perera, 2020) estudiaron la problemática de la separación de datos durante la transición de sistemas monolíticos hacia arquitecturas distribuidas. Sus resultados evidencian que la definición incorrecta de los límites de persistencia puede producir redundancia de información, incremento en la complejidad transaccional y nuevas dependencias entre servicios. Asimismo, destaca que la mayoría de los enfoques existentes continúan dependiendo de decisiones manuales realizadas por expertos del dominio, lo que limita la reproducibilidad y automatización del proceso de migración.

Desde una perspectiva orientada a la persistencia distribuida, (Munonye & Martinek, 2020) evalúan diferentes patrones de almacenamiento utilizados en arquitecturas de microservicios, entre ellos Database per Service, CQRS y Event Sourcing. Los autores concluyen que la separación de los datos favorece la independencia operativa de los servicios y reduce el acoplamiento entre componentes. Sin embargo, reconocen que estos patrones asumen la existencia previa de límites de servicio claramente definidos, por lo que no proporcionan mecanismos para determinar qué entidades deben formar parte de cada fragmento de datos.

Una problemática similar es identificada por (Kazanavičius et al., 2022), quienes analizan la migración de bases de datos monolíticas hacia esquemas de persistencia distribuidos basados en una persistencia polígota (Polyglot Persistence). Su propuesta demuestra que diferentes tecnologías de almacenamiento pueden coexistir dentro de una misma arquitectura de microservicios, permitiendo optimizar el rendimiento y la escalabilidad de cada servicio de manera independiente. Los autores reconocen que la principal dificultad no radica en seleccionar la tecnología de almacenamiento adecuada, sino en determinar previamente la distribución correcta de las entidades de datos entre los servicios resultantes.

De forma complementaria, diversos enfoques basados en Domain-Driven Design y los descritos en (Gysel et al., 2016) consideran las dependencias de datos como uno de los criterios para delimitar servicios. Sin embargo, estos métodos utilizan la información de persistencia únicamente como un elemento adicional dentro del proceso de identificación de microservicios y no como el objetivo principal de la descomposición. Como resultado, la descomposición de la base de datos suele derivarse de los límites de servicio obtenidos, en lugar de ser analizada de manera explícita y sistemática.

Los estudios de mapeo sistemático más recientes confirman esta tendencia. (Martínez et al., 2025) reportan que la mayoría de las investigaciones se concentra en la identificación de candidatos a microservicios mediante análisis estructural, dinámico o de dominio, mientras que la descomposición de la base de datos recibe una considerablemente menor atención. De manera similar, (Carvalho et al., 2020) observan que gran parte de los enfoques existentes consideran únicamente dependencias de software y relaciones funcionales atendiendo solo la cohesión y el acoplamiento, dejando fuera aspectos relacionados con la persistencia y las interacciones entre entidades de datos.

En conclusión, aunque existen múltiples propuestas relacionadas con persistencia distribuida, patrones de almacenamiento y administración de datos en arquitecturas de microservicios, la literatura aún carece de métodos que permitan identificar de forma sistemática las agrupaciones de entidades de datos asociadas a una capacidad de negocio específica durante la migración de sistemas monolíticos. Esta brecha resulta relevante debido a que la correcta descomposición de la base de datos constituye un requisito fundamental para cumplir con las características y buenas prácticas al implementar una arquitectura de microservicios.

Identificación de Microservicios y Delimitación de Servicios

La identificación de candidatos a microservicios constituye una de las áreas de investigación más activas dentro de la modernización de sistemas monolíticos. La definición adecuada de los límites de servicio influye directamente en atributos arquitectónicos como la cohesión, el acoplamiento, la mantenibilidad y la autonomía de los microservicios resultantes. Debido a ello, la comunidad científica ha propuesto múltiples enfoques para apoyar la delimitación de servicios a partir de diferentes fuentes de información, incluyendo el código fuente, los modelos de dominio, las trazas de ejecución y las dependencias existentes entre componentes.

Los primeros trabajos se orientan principalmente al análisis estructural de sistemas monolíticos. En esta línea, (Levcovitz et al., 2016) y (Mazlami et al., 2017) proponen mecanismos para identificar agrupaciones funcionales mediante el análisis de dependencias entre módulos, clases y componentes. Estos enfoques parten de la premisa de que los elementos con alta cohesión funcional y bajo acoplamiento externo constituyen candidatos naturales a microservicios. Los resultados obtenidos demuestran la utilidad del análisis estático para apoyar la descomposición arquitectónica; sin embargo, las propuestas se concentran fundamentalmente en la estructura del software y no consideran de manera explícita las relaciones existentes con los modelos de datos.

Posteriormente, diversos investigadores incorporan información dinámica con el propósito de capturar el comportamiento real del sistema durante su ejecución. (Krause et al., 2020) proponen una estrategia que combina análisis estático y dinámico para identificar límites de servicio mediante el estudio simultáneo de dependencias estructurales y trazas de ejecución. De forma similar, (Matias et al., 2020) y (Andrade et al., 2023) utilizan información obtenida a partir de la ejecución del sistema para mejorar la precisión de los agrupamientos identificados. Estos trabajos evidencian que las interacciones observadas durante la operación del sistema permiten comprender mejor las dependencias funcionales entre componentes. No obstante, el análisis continúa centrándose en la delimitación de servicios y no en la identificación de las entidades de datos que participan en dichas interacciones.

Otra corriente de investigación se fundamenta en el análisis del dominio de negocio. (Baresi et al., 2017) proponen identificar microservicios mediante el estudio de interfaces y funcionalidades expuestas por el sistema, mientras que los enfoques basados en Domain-Driven Design utilizan conceptos como Bounded Context y Ubiquitous Language para delimitar responsabilidades de negocio. Desde esta perspectiva, los límites de servicio se derivan de las capacidades organizacionales y de los procesos que generan valor dentro de la empresa. Este enfoque ha demostrado ser especialmente útil para garantizar la alineación entre la arquitectura de software y el dominio de negocio. Sin embargo, la delimitación obtenida depende significativamente del conocimiento de expertos y de actividades de análisis en su mayoría manuales.

Con el objetivo de reducir la intervención humana, investigaciones más recientes han explorado enfoques semiautomáticos y automáticos para la identificación de microservicios. (A. Selmadji et al., 2020) propusieron un enfoque para la identificación de microservicios durante la migración de sistemas monolíticos, basado en el análisis conjunto de las relaciones entre los elementos del código fuente y los datos persistentes manipulados por dichos elementos. Además, la propuesta incorpora el conocimiento del arquitecto para orientar el proceso de identificación y define una función de calidad que evalúa la cohesión e interdependencia de los microservicios obtenidos. Los resultados experimentales reportados por los autores muestran que la utilización simultánea de información estructural y de persistencia permite obtener agrupaciones de servicios con un mayor grado de consistencia funcional. Sin embargo, la propuesta se orienta principalmente a la identificación de microservicios y no aborda la descomposición de la base de datos ni la generación de límites de persistencia bajo el enfoque de una base de datos por servicio.

A pesar de las diferencias existentes entre los enfoques propuestos, la mayoría comparte el objetivo común de identificar agrupaciones funcionales que puedan evolucionar de manera independiente dentro de una arquitectura de microservicios. Como consecuencia, los criterios de evaluación utilizados suelen estar relacionados con aspectos como acoplamiento, granulación y alineación con procesos de negocio. Sin embargo, las entidades de datos involucradas en la ejecución de dichas funcionalidades reciben una atención considerablemente menor dentro de los procesos de identificación.

Esta situación es señalada por diversos estudios de mapeo sistemático. (Hassan et al., 2024) y (Martínez et al., 2025) concluyen que la literatura ha concentrado gran parte de sus esfuerzos en determinar cómo identificar límites funcionales de servicio, mientras que la relación entre dichos límites y la descomposición de la base de datos permanece insuficientemente explorada.

A partir del análisis de los trabajos revisados se observa que la identificación de microservicios ha evolucionado desde enfoques puramente estructurales hacia propuestas que incorporan información dinámica y conocimiento del dominio. Pero la fragmentación de la base de datos generalmente se aborda como una actividad posterior e independiente del proceso de identificación de servicios.

En conclusión, la literatura proporciona múltiples mecanismos para delimitar servicios y apoyar la migración hacia arquitecturas de microservicios, pero aún existe la necesidad de enfoques que integren de manera simultánea el análisis de capacidades de negocio, interacciones entre componentes y entidades de datos. Esta integración permitiría obtener no sólo candidatos a microservicios, sino también las agrupaciones de datos asociadas a cada servicio, contribuyendo a una migración más consistente y una reducción efectiva del acoplamiento indirecto derivado de la persistencia compartida. A continuación, en la tabla 8 se presenta un resumen sobre los trabajos relacionados con esta investigación, atendiendo las características de migración, identificación de los servicios, el tipo de análisis, así como la atención del modelo de datos y la secuencia de interacciones.

Tabla 8 Resumen de artículos sobre trabajos relacionados con esta investigación.

Trabajo	Migración de Monolitos	Identifica Microservicios	Análisis Estático	Análisis Dinámico	Modelo de Datos	Sec. de Interacción
(Levcovitz et al., 2016)	✓	✓	✓	X	X	X
(Mazlami et al., 2017)	✓	✓	✓	X	X	X
(Baresi et al., 2017)	X	✓	X	X	X	X
(Furda et al., 2017)	✓	✓	✓	X	-	X
(Hernández-Ortega et al., 2017)	✓	-	X	X	X	X
(A. Selmadji et al., 2020)	✓	✓	✓	X	X	X
(Krause et al., 2020)	✓	✓	✓	✓	-	✓
(Munonye & Martinek, 2020)	X	X	X	X	✓	X
(Matias et al., 2020)	✓	✓	✓	✓	X	✓
(Bandara & Perera, 2020)	✓	-	✓	-	✓	X

Trabajo	Migración de Monolitos	Identifica Microservicios	Análisis Estático	Análisis Dinámico	Modelo de Datos	Sec. de Interacción
(Gaffney et al., 2021)	X	X	X	X	✓	X
(Andrade et al., 2023)	✓	✓	✓	✓	-	✓
(Kazanavičius et al., 2022)	✓	-	X	X	✓	X
(Zuckmantel et al., 2022)	X	X	X	X	✓	X
(Fritzsche et al., 2024)	✓	-	X	X	X	X
(Taibi et al., 2020)	✓	-	X	X	X	X
Esta tesis	✓	✓	✓	X	✓	✓

Donde:

✓ = Soporte explícito.

- = Soporte parcial.

X = No considerado.

Del análisis de los trabajos relacionados, en general, se concluye que, si bien el estudio sobre la arquitectura de microservicios no es un tema reciente, aún persisten desafíos significativos que continúan siendo objeto de investigación. Entre los principales retos identificados destacan la separación de los datos y la preservación de la consistencia de la información al momento de extraer microservicios desde un sistema monolítico o legado.

Se observó una gran cantidad de investigaciones enfocadas en la migración de sistemas monolíticos hacia arquitecturas de microservicios, muchas de las cuales proponen metodologías diversas para llevar a cabo dicho proceso. No obstante, se detectó una escasa atención específica al problema de la descomposición o separación de la base de datos monolítica durante la migración. Esta limitación podría deberse a que la separación de datos representa una tarea compleja, incluso para equipos de desarrollo con experiencia, ya que implica un análisis detallado y profundo para garantizar la integridad, autonomía y coherencia de las nuevas bases de datos por servicio.

Como resultado de esta revisión, se identificó una oportunidad clara para aportar a la investigación en este campo, particularmente en lo referente al desarrollo de métodos que faciliten la descomposición efectiva de bases de datos monolíticas, de sistemas legados, hacia esquemas distribuidos, alineados con una arquitectura de microservicios. Este trabajo de investigación sobresale de los demás trabajos relacionados debido a que el método mediante reglas genera microservicios y de manera paralela genera la base de datos, además, propone el patrón observador para manejar la consistencia de información lo que otros trabajos relacionados no hacen de forma integral.

2.6.2 Trabajos internos del laboratorio de ingeniería de software

En el CENIDET se han realizado proyectos de reingeniería para mejorar la arquitectura de software. Como antecedentes a esta investigación se desarrollaron trabajos de tesis que tratan de manera similar el problema, pero con diferentes enfoques de solución. Estos son:

- “Generación de Servicios Web desde Marcos Orientados a Objetos a partir de Clases Colaborativas en Casos de Uso (MOOaSW)” (Pérez, 2009)
- “Transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con Arquitectura Orientada a Servicios (INLineCLASS)” (Legorreta, 2017).
- “Generación de Servicios Web desde Marcos Orientados a Objetos con el Enfoque de Clases Internas (MOOinMS)” (Gallardo, 2017).
- “Método para migrar servicios web bajo arquitectura SOA hacia Microservicios con arquitectura REST (MSWeb2MMServ)” (Hernandez, 2018).

Los cuatro enfoques consideran el caso de uso como un componente reusable ya que cada uno satisface una capacidad o requerimiento de un dominio o negocio. De este modo, cada componente reusable identificado es implementado en un caso de uso del Marco Orientado a Objetos original (MOO) y envuelto en una capa SOA o REST para implementar y desplegar los Servicios que integran al Marco de Servicios de salida.

En el trabajo de (Pérez, 2009), cada caso de uso respeta las relaciones arquitecturales entre clases como fueron definidas en el MOO original. Así la arquitectura interna de clases resultante para cada servicio web se organiza con el conjunto de clases relacionadas con sus funciones y datos que son utilizados en la secuencia interactiva del Caso de Uso. En la tesis, *“Transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con Arquitectura Orientada a Servicios”*, se utiliza un método de análisis estático que rastrea totalmente el código del Marco Orientado a Objetos (MOO) siguiendo las reglas gramaticales del lenguaje java, considerando que éste es el lenguaje en el que el código nativo del MOO original está implementado. Cada componente definido en un Servicio Web implementa un caso de uso (CU) del MOO original. Otro rasgo de este enfoque radica en que la metodología usada permite descubrir los CU utilizados por actores externos que saben de las capacidades que deben soportar los procesos de negocios, así como por actores lógicos internos, que son unidades de programa u otros servicios web que interactúan e intercambian información para producir una aplicación de software completa. La arquitectura de cada servicio web está integrada en una única clase envolvente, la cual contiene la totalidad de las funciones y datos que participan en la secuencia interactiva de un caso de uso.

El enfoque del trabajo de tesis referido, *“Generación de Servicios Web desde Marcos Orientados a Objetos con el Enfoque de Clases Internas”*, consiste en mejorar la modularidad de los Servicios Web que conforman Marcos de Servicios Web con arquitectura SOA, mediante

la composición de clases internas y el agrupamiento racional de funciones que participan en secuencias interactivas de casos de uso, así como los datos que son manipulados por estas funciones, cuyo objetivo es obtener servicios web con niveles balanceados de coherencia, cohesión, factor de acoplamiento y el tiempo de respuesta en su arquitectura interna, de tal manera que se logre un balance entre estos atributos de calidad, para obtener un mejor desempeño y mejores condiciones de reuso y mantenimiento. El producto resultante de este proyecto de tesis es implementado en una extensión a la herramienta SR2-Transforming que da soporte a la Reingeniería de Marcos de Aplicaciones Orientados a Objetos hacia Marcos de Servicios equivalentes.

El cuarto enfoque del trabajo de tesis referido, *“Método para migrar servicios web bajo arquitectura SOA hacia Microservicios con arquitectura REST”* consiste en un método implementado en una herramienta de transformación para descomponer de manera semiautomática Marcos Orientados a Objetos, los cuales, por tener un alto grado de integración son altamente acoplados. La transformación se realiza identificando el código de servicios integrados en los MOOs de entrada y separándolos para su desacoplamiento, a través de su transformación en microservicios REST. La identificación y descomposición se realiza con base en cuatro enfoques que mejoran los atributos de calidad como son la coherencia, cohesión, factor de acoplamiento y tiempo de respuesta. Los microservicios obtenidos logran un rendimiento mejorado con base en su protocolo de comunicación (REST bajo HTTP) contra sus equivalentes Servicios Web bajo el protocolo SOAP.

Los enfoques mencionados obtienen, respectivamente, servicios web o microservicios cumpliendo con su objetivo, pero ninguno atiende la separación de la base de datos, es decir, que la base de datos se mantiene intacta. Cuando varias entidades de software acceden a datos globales, ya sea que éstos sean declarados como variables globales, estáticas o dispuestas en una base de datos global, se produce un acoplamiento indirecto entre estas entidades; el acoplamiento produce dependencias y las dependencias producen fragilidad ante cambios, lo que representa un problema al momento de implementar el despliegue de los mismos, ya que cada microservicio atiende una meta de valor y debido a que varios microservicios pueden utilizar ya sea de lectura o escritura las mismas tablas de la base de datos, puede haber inconsistencia e incluso pérdida de información al momento de desplegarlos. En (Dragoni, 2017) se menciona que, cuando se tienen pequeños fragmentos de código con una única base de datos, la cual es compartida por todos estos fragmentos, no se cumple con la propiedad de aislamiento de la base de datos; éstos son conocidos como microlitos. Por definición, los microservicios deberían ser pequeñas piezas de código con autonomía, cohesión, coherencia, ocultamiento de información y aislamiento.

Capítulo 3: Materiales y Métodos

3.1 Descripción general del Método MONO2MSDB

La transición de un sistema monolítico a una arquitectura de microservicios implica la división de un sistema centralizado en componentes distribuidos e independientes. Esta descomposición presenta un desafío significativo cuando se trata de separar la base de datos de un sistema monolítico en bases de datos específicas para cada microservicio. En este apartado, se describe el método para identificar y descomponer la base de datos monolítica en múltiples bases de datos, cada una asociada a un microservicio en particular, mediante el análisis estático del código fuente y mediante la identificación del modelo de datos asociado a una secuencia interactiva de un caso de uso que atiende una meta de valor y representa un candidato a microservicio.

El método propuesto en esta investigación constituye un procedimiento sistemático para la identificación, delimitación y generación de microservicios candidatos a partir del código fuente de sistemas de software monolíticos. El método es tecnológicamente agnóstico en su formulación conceptual: puede implementarse sobre cualquier sistema monolítico que exhiba una estructura de capas bien definida con separación entre la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos, independientemente del lenguaje de programación, el framework o el mecanismo de persistencia utilizados. Esta generalidad es una propiedad del diseño metodológico, que separa los principios del método de los detalles de implementación de una herramienta específica.

El método opera exclusivamente sobre el código fuente sin requerir la ejecución del sistema bajo análisis, característica que lo enmarca en la categoría de técnicas de análisis estático y que garantiza su aplicabilidad en etapas tempranas del proceso de migración. El único requisito de entrada es que el código fuente del sistema monolítico esté disponible, libre de errores de compilación y que la arquitectura respete una separación identificable entre sus capas funcionales.

La contribución central del método reside en tres elementos diferenciadores respecto a los enfoques previos de la literatura: (1) la adopción de la secuencia interactiva de caso de uso definida como la traza de ejecución completa desde un punto de entrada de la capa de presentación hasta las entidades del dominio que son accedidas como unidad de descomposición; (2) la formulación de un conjunto de reglas formales para la separación del modelo de datos entre los microservicios resultantes, abordando el problema de las entidades compartidas mediante duplicación controlada y consistencia eventual sugiriendo el patrón observador para atender esta problemática; y (3) la definición de la métrica Índice de Acoplamiento entre tablas compartidas (IATC) como instrumento de evaluación cuantitativa del beneficio arquitectónico de la descomposición desarrollada.

El método se estructura en seis fases secuenciales: (1) análisis sintáctico del código fuente mediante extracción de árbol de sintaxis abstracta (AST); (2) identificación y clasificación de componentes según su estrato arquitectónico (mapeo de los elementos identificados); (3) resolución transversal de la cadena de dependencias entre componentes; (4) resolución transitiva de relaciones entre entidades del dominio; (5) aplicación del conjunto de reglas de separación del modelo de datos distribuido; y (6) generación de los artefactos del microservicio. La Figura 3 presenta el diagrama BPMN principal de forma general sobre el proceso de migración y descomposición de la base de datos de un sistema monolítico para operar en una arquitectura de microservicios.

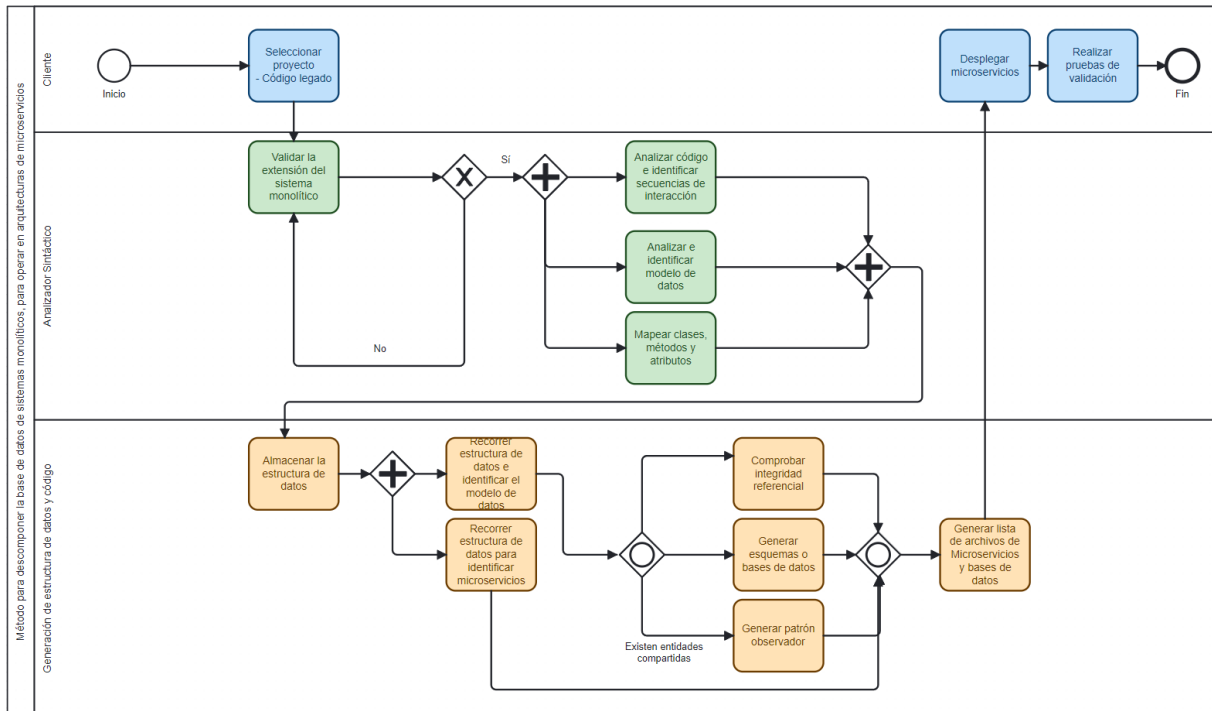


Figura 3 Proceso BPMN del método para descomponer la BD de un sistema monolítico hacia microservicios.

3.2 Fundamentos Conceptuales del Método

3.2.1 Arquitectura en Capas como Precondición del Método

El método presupone que el sistema monolítico bajo análisis implementa una arquitectura en capas en la que es posible distinguir, al menos, los siguientes tres estratos con responsabilidades claramente diferenciadas:

- La capa de presentación o interfaz que constituye el punto de entrada del sistema monolítico y es responsable de recibir las solicitudes provenientes de usuarios u otros sistemas para canalizarlas hacia la lógica de negocio. Dependiendo de la tecnología

utilizada por la aplicación, esta capa puede implementarse mediante interfaces web, anotaciones como `@controller`, sin que ello implique una arquitectura de microservicios, ya que todas estas interfaces forman parte de una única aplicación desplegada sobre una base de datos compartida.

- La capa de lógica de negocio o de dominio, que contiene los componentes responsables de implementar las reglas de negocio del sistema, orquestar las operaciones y coordinar el flujo de datos entre la presentación y el acceso a datos. Los componentes de esta capa implementan el comportamiento central del sistema.
- La capa de acceso a datos o de persistencia, que contiene los componentes responsables de interactuar con el almacén de datos persistente del sistema (base de datos relacional, base de datos documental, sistema de archivos, servicio externo, u otros). Estos componentes encapsulan las operaciones de creación, lectura, actualización y eliminación (CRUD) sobre las entidades del dominio.

La existencia de esta separación en capas, que no requiere ser rigurosa sino suficientemente identificable en el código fuente, es la condición necesaria que permite al método trazar la cadena de dependencias desde los puntos de entrada hasta las entidades del dominio y, a partir de ella, delimitar los microservicios candidatos con sus respectivos modelos de datos.

Cuando el sistema bajo análisis no implementa una separación en capas explícita, por ejemplo, en sistemas con lógica de negocio entremezclada con el acceso a datos, o en sistemas con una arquitectura de script procedural, el método puede igualmente identificar candidatos a través de las reglas de identificación de componentes por convención de nomenclatura, aunque con menor precisión respecto a la delimitación de responsabilidades entre microservicios.

3.2.2 La Secuencia Interactiva de Casos de Uso como unidad de Descomposición

La decisión metodológica más significativa del método desarrollado consiste en emplear la secuencia de interacción de un caso de uso como unidad atómica para la descomposición del monolito en microservicios. Esta elección se fundamenta en el principio de alineación entre la granularidad de los microservicios y la granularidad de las operaciones de negocio observables desde el exterior del sistema.

Un componente de entrada (controlador, handler, endpoint) de un sistema monolítico expone típicamente múltiples operaciones de negocio agrupadas por afinidad semántica. Cada una de estas operaciones representada por un método o función del componente de entrada constituye una secuencia interactiva de caso de uso: una traza de ejecución que comienza en el punto de entrada, recorre los componentes de lógica de negocio que dicho punto invoca, y concluye en las entidades del dominio que los componentes de acceso a datos recuperan o modifican durante esa operación específica.

Formalmente, dado un componente de entrada C con operación o , la secuencia interactiva asociada se define como la tupla:

$$SI(o) = (o, \{L_1, L_2, \dots, L_k\}, \{A_1, A_2, \dots, A_n\}, \{E_1, E_2, \dots, E_p\})$$

donde $\{L_1, \dots, L_k\}$ es el conjunto de componentes de lógica de negocio (caso de uso) invocados directa o indirectamente desde o ; $\{A_1, \dots, A_n\}$ es el conjunto de componentes de acceso a datos utilizados por esos componentes de lógica de negocio; y $\{E_1, \dots, E_p\}$ es el conjunto de entidades del dominio gestionadas por esos componentes de acceso a datos, incluyendo las entidades alcanzables transitivamente a través de las relaciones declaradas entre entidades.

Cada secuencia interactiva $SI(o)$ identificada en el monolito da origen a un microservicio candidato cuya responsabilidad funcional es ejecutar exclusivamente la operación de negocio que representa un caso de uso. El nombre del microservicio se deriva del nombre de la operación en notación PascalCase, y el conjunto de entidades $\{E_1, \dots, E_p\}$ determina el modelo de datos que el microservicio debe poseer en su propia base de datos.

La granularidad resultante de esta elección es de grano fino, un microservicio por operación de negocio observable y que está alineado con el principio de responsabilidad única y con el patrón de diseño de microservicios de grano fino (fine-grained microservices) (Fowler, 2002). En sistemas donde esta granularidad resulte excesiva, los microservicios generados pueden agruparse manualmente según criterios semánticos o de contexto delimitado (Bounded Context) propios del diseño dirigido por dominio.

3.3 Fase 1: Análisis Sintáctico del Código Fuente

La primera fase del método transforma el código fuente del sistema monolítico en una representación estructurada que permite extraer la información arquitectónica necesaria para las fases subsecuentes. La representación adoptada es el árbol de sintaxis abstracta (AST, por sus siglas en inglés Abstract Syntax Tree), que captura la estructura sintáctica del código de forma jerárquica e independiente del formato textual concreto.

El proceso de construcción del AST sigue tres etapas estándar: (i) análisis léxico, que segmenta el flujo de caracteres del código fuente en unidades léxicas (tokens) identificadores, palabras reservadas, operadores, literales, delimitadores; (ii) análisis sintáctico, que organiza los tokens en un árbol de acuerdo con la gramática del lenguaje de programación empleado; y (iii) recorrido del AST, que extrae la información de interés mediante visitas selectivas a los nodos del árbol correspondientes a declaraciones de clase, declaraciones de campo, declaraciones de método e interfaces implementadas.

El método utiliza ANTLR4 (Another Tool for Language Recognition por sus siglas en inglés) como motor de análisis sintáctico, y en su formulación genérica es independiente del parser concreto: para otros lenguajes de programación, el mismo proceso puede realizarse con parsers equivalentes (Roslyn para C#/.NET, ast para Python, ts-morph para TypeScript, RubyParser

para Ruby), dado que todos ellos producen una representación AST sobre la que puede aplicarse el mismo algoritmo de extracción.

Un aspecto crítico de robustez del análisis es que el recorrido del AST se ejecuta sobre la totalidad de los archivos de código fuente de la aplicación en un único paso acumulativo, de modo que la información extraída de un archivo (por ejemplo, los componentes de acceso a datos que un componente de lógica de negocio inyecta) esté disponible al momento de procesar otro archivo que hace referencia a ese componente. Este diseño de un solo proceso acumulativo es esencial para la resolución transversal de dependencias descrita en la Fase 3.

3.4 Fase 2: Identificación de Componentes por Estrato Arquitectónico

En esta fase se clasifica cada componente encontrado en el código fuente según el estrato arquitectónico al que pertenece: componente de entrada, componente de lógica de negocio, componente de acceso a datos, o entidad del dominio. Esta clasificación es el fundamento sobre el que se construye la cadena de dependencias de cada secuencia interactiva de caso de uso.

El método define dos estrategias de clasificación aplicadas en orden de precedencia, lo que garantiza una cobertura amplia ante diferentes estilos de codificación:

Estrategia 1: Clasificación por metadatos explícitos

En frameworks modernos, los componentes declaran su rol arquitectónico mediante mecanismos de anotación, atributos o decoradores que pueden extraerse directamente del AST. La Tabla 9 presenta la correspondencia entre los metadatos explícitos y el estrato arquitectónico al que atienden con ejemplos representativos de los frameworks más utilizados en la actualidad.

Tabla 9 Estrategias de clasificación de componentes por metadatos explícitos.

Estrato	Ejemplos de metadatos explícitos	Frameworks representativos
Entrada / Presentación	@Controller, @RestController, @RequestMapping, @GetMapping, [ApiController], [Route], @app.route, @Controller()	Spring MVC (Java), ASP.NET Core (C#), Flask/FastAPI (Python), Express.js (Node.js), Ruby on Rails, Laravel (PHP)
Lógica de negocio	@Service, @Component, @ApplicationService, [Service], @Injectable() con sufijo Service	Spring (Java), ASP.NET Core (C#), Angular/NestJS (TS), CDI (Jakarta EE)
Acceso a datos	@Repository, @DAO, extends JpaRepository / Repository<T> / DbContext, @app.route con ORM, implements IRepository	Spring Data (Java), EF Core (C#), SQLAlchemy (Python), ActiveRecord (Ruby), Eloquent (PHP)
Entidad del dominio	@Entity, @Table, @Document, @Model, [Table], class ...Model, DbSet<T>	JPA/Hibernate (Java), EF Core (C#), Mongoose (Node.js), Django Models (Python), ActiveRecord (Ruby)

Estrategia 2: Clasificación por convención de nomenclatura

Cuando el código fuente no utiliza metadatos explícitos o cuando el mecanismo de anotación/atributo no es identificado durante el análisis el método, se aplica una clasificación por convención de nomenclatura, reconociendo patrones de sufijo ampliamente adoptados en la industria del software:

- Componentes de entrada: Sufijos Controller, Handler, Resource, Endpoint, Router, Presenter.
- Componentes de lógica de negocio: Sufijos Service, ServiceImpl, Manager, Facade, UseCase, Interactor, ApplicationService.
- Componentes de acceso a datos: sufijos Repository, RepositoryImpl, DAO, Dao, DataAccess, Store, Mapper, Gateway (cuando actúa como repositorio).
- Entidades del dominio: clases que coinciden con el tipo genérico declarado en la super interfaz del componente de acceso a datos, o clases cuyos nombres aparecen como tipos de los campos de acceso a datos.

Esta estrategia actúa como el mecanismo que garantiza la continuidad del método (fallback) y permite que el método sea aplicable a códigos de base de datos que siguen convenciones, pero carecen de anotaciones, como algunas aplicaciones legadas desarrolladas antes de la adopción generalizada de frameworks de inyección de dependencias.

Reglas de Identificación de Secuencias Interactivas de Caso de Uso

Una vez clasificados los componentes, el método aplica las reglas descritas en la tabla 10 para identificar qué operaciones de los componentes de entrada constituyen secuencias interactivas de casos de uso:

Tabla 10 Reglas de identificación de secuencias interactivas de caso de uso.

Regla	Nombre	Descripción
R-01	Visibilidad pública	Solo las operaciones con visibilidad pública o equivalente (public, exported, accesibles desde el exterior del módulo) son candidatas a secuencia interactiva. Las operaciones privadas, protegidas o internas representan lógica auxiliar, no puntos de entrada del negocio.
R-02	Exclusión de configuración del framework	Las operaciones cuya semántica es configurar el framework en lugar de implementar lógica de negocio se excluyen. Esto incluye métodos de inicialización del ciclo de vida del framework, manejadores de errores globales, configuradores de serialización/validación, y operaciones de preprocesamiento de solicitudes.
R-03	Exclusión de operaciones sin acceso a datos	Las operaciones que no trazan ninguna ruta hacia componentes de acceso a datos directa ni indirectamente pueden excluirse del conjunto de candidatos, dado que no

Regla	Nombre	Descripción
		gestionan estado persistente y no generan un modelo de datos propio. Su funcionalidad puede preservarse como lógica interna o como función utilitaria.
R-04	Nombre derivado de la operación	El nombre del microservicio candidato, se deriva del nombre de la operación del componente de entrada, normalizado en PascalCase y con el sufijo Service. Esto garantiza trazabilidad directa entre la operación original del monolito y el microservicio generado.
R-05	Análisis por operación, no por componente	La identificación de entidades del dominio se realiza por operación individual, no por el componente de entrada completo. Dos operaciones del mismo componente que acceden a entidades distintas generan dos candidatos a microservicio distintos, con sus propios modelos de datos, evitando que todas las entidades del componente sean atribuidas indiscriminadamente a todos sus métodos.

3.5 Fase 3: Resolución transversal en la secuencia de interacción

La cadena de dependencias de una secuencia interactiva desde el componente de entrada hasta las entidades del dominio no puede resolverse dentro de un único archivo de código fuente cuando los componentes están distribuidos en múltiples archivos, como es habitual en sistemas monolíticos con cientos o miles de clases. En esta fase se resuelve esta cadena de forma transversal sobre el conjunto de mapas acumulados durante la Fase 2.

El proceso de resolución sigue tres niveles de la cadena de dependencias:

Nivel 1: Componente de entrada a componente de lógica de negocio: para cada operación del componente de entrada, el método examina los campos inyectados en el componente cuyo tipo corresponde a un componente de lógica de negocio, y adicionalmente analiza el cuerpo de la operación en busca de invocaciones sobre esos campos. Únicamente los componentes de la lógica de negocio invocados por la operación específica se asocian al candidato correspondiente.

Nivel 2: Componente de lógica de negocio a componente de acceso a datos: para cada componente de lógica de negocio asociado al candidato, el método extrae los campos inyectados cuyo tipo corresponde a un componente de acceso a datos. Todos los componentes de acceso a datos inyectados en un componente de lógica de negocio son considerados potencialmente utilizados por cualquier operación que invoque ese componente.

Nivel 3: Componente de acceso a datos a entidad del dominio: para cada componente de acceso a datos identificado, el método extrae la entidad del dominio que gestiona, a partir de la declaración del tipo genérico en la interfaz extendida (por ejemplo, `Repository<Producto>`, `DbContext` con `DbSet<Producto>`, o análogo). Esta entidad se agrega al conjunto de entidades del candidato.

El método contempla además el patrón de acceso directo a componentes de acceso a datos desde el componente de entrada, sin intermediación de un componente de lógica de negocio. En este caso, la resolución omite el Nivel 2 y procede directamente del Nivel 1 al Nivel 3, aplicando la Regla R-05 para garantizar que las entidades sean atribuidas a la operación específica que las invoca, no al componente de entrada completo.

Para tratar el patrón Interface, ampliamente utilizado en arquitecturas que implementan el principio de inversión de dependencias, donde el componente de entrada declara un campo del tipo interfaz IServicioX mientras la implementación real es la clase ServicioXImpl, el método aplica una normalización de nombres. Este proceso elimina el prefijo convencional de interfaz (I, I_) y el sufijo convencional de implementación (Impl, Implementation) antes de realizar la comparación, garantizando que la interfaz y su implementación sean tratadas como el mismo componente durante la resolución.

3.6 Fase 4: Resolución Transitiva de Relaciones entre Entidades del Dominio

Las entidades del dominio de un sistema monolítico establecen frecuentemente relaciones entre sí a través de mecanismos de asociación declarados en el código fuente: claves foráneas en los campos de la entidad, relaciones uno-a-muchos, muchos-a-uno, muchos-a-muchos, u otras formas de referencia entre tipos de entidad. Cuando una operación de negocio accede a una entidad que tiene relaciones declaradas con otras entidades, el correcto funcionamiento de esa operación puede requerir el acceso implícito o explícito a las entidades relacionadas.

En esta fase se aplica un algoritmo de cierre transitivo sobre el grafo de relaciones entre entidades para agregar al conjunto de entidades de cada candidato todas las entidades alcanzables a través de las relaciones declaradas, partiendo de las entidades identificadas en la Fase 3. El algoritmo opera de la siguiente manera:

1. Para cada entidad E en el conjunto de entidades del candidato, se examina el conjunto de campos de E que referencian otras entidades del dominio.
2. Para cada campo de tipo entidad F referenciado en E, si F pertenece al conjunto de entidades conocidas del sistema y aún no ha sido incluida en el candidato, F se agrega al conjunto de entidades del candidato.
3. Los pasos anteriores se repiten hasta que ninguna entidad nueva es agregada en un ciclo completo (convergencia del cierre transitivo).

El método identifica dos patrones de referencia entre entidades que activan la resolución transitiva:

Referencia directa: el tipo del campo es directamente el nombre de otra entidad del dominio (por ejemplo, un campo pedido de tipo Pedido en la entidad LineaDetalle).

Referencia mediante colección genérica: el tipo del campo es una colección parametrizada con el nombre de otra entidad (por ejemplo, un campo líneas de tipo `List<LineaDetalle>`, `Collection<LineaDetalle>` o análogo en el lenguaje de programación empleado).

La resolución transitiva garantiza que el modelo de datos asignado a cada microservicio candidato sea completo y operacionalmente autosuficiente: ninguna entidad requerida para la ejecución de la operación de negocio queda fuera del conjunto de entidades del candidato por no haber sido accedida directamente a través de un componente de acceso a datos.

3.7 Fase 5: Separación del Modelo de Datos Distribuido

Esta fase constituye el núcleo metodológico diferenciador del método desarrollado respecto a enfoques previos, y aborda el problema más complejo de la descomposición de sistemas monolíticos: la separación del modelo de datos centralizado en modelos de datos aislados, uno por cada microservicio, manteniendo la consistencia de la información distribuida. Esta fase se estructura en torno a un conjunto de reglas formales que determinan la propiedad de las entidades, la estrategia de replicación y el mecanismo de sincronización.

Principio Fundamental: Base de Datos por Microservicio

El principio arquitectónico sobre el que se fundamenta esta fase es el denominado principio de base de datos por microservicio (*Database per service*), que establece que cada microservicio debe poseer y gestionar exclusivamente su propio almacén de datos, sin compartir el esquema ni la instancia de base de datos con ningún otro microservicio. Este principio es condición necesaria para garantizar las propiedades de independencia de despliegue, autonomía operacional y resiliencia ante fallos que caracterizan la arquitectura de microservicios.

El cumplimiento estricto de este principio implica que, cuando dos o más operaciones de negocio del monolito acceden a la misma entidad del dominio, no es posible simplemente descomponer el monolito asignando esa entidad a un único microservicio. De hacerlo, los demás microservicios que la requieran establecerán dependencias de datos sobre un tercero, lo que recrea el acoplamiento estructural que la migración pretende eliminar. La solución adoptada por el método consiste en la duplicación controlada de entidades con sincronización por eventos, que es el objeto central de las reglas que se presentan en las subsecciones siguientes.

Reglas de Separación del Modelo de Datos

El método define nueve reglas formales de separación del modelo de datos, organizadas en tres grupos: reglas de detección de compartición, reglas de asignación de propiedad, y reglas de replicación y consistencia, estas son descritas en la tabla 11.

Tabla 11 Reglas formales de separación del modelo de datos distribuido.

Regla	Nombre	Descripción formal
Grupo I		
Reglas de detección de compartición de entidades		
R-DB-1	Detección de entidad compartida	Una entidad E es compartida si y sólo si el número de candidatos distintos cuyo conjunto de entidades contiene a E es mayor o igual a 2: $ \{c \in \text{Candidatos} : E \in \text{entidades}(c)\} \geq 2$. Toda entidad compartida activa el proceso de replicación definido en el Grupo III.
R-DB-2	Detección de compartición transitiva	Si una entidad E es compartida y E tiene una relación declarada con otra entidad F, entonces F debe evaluarse también como posible entidad compartida, incluso si F no aparece directamente en el conjunto de entidades de múltiples candidatos. La compartición puede originarse en la relación, no solo en el acceso directo.
Grupo II		
Reglas de asignación de propiedad de entidades		
R-DB-3	Propiedad por operación de escritura	El microservicio propietario de una entidad E es aquel cuya operación de negocio crea o modifica registros de E (operaciones CREATE, UPDATE o DELETE sobre el almacén de E). Si múltiples candidatos realizan escrituras sobre E, se designa como propietario al candidato cuya operación de negocio tiene mayor semántica de gestión directa sobre E (típicamente la operación de creación).
R-DB-4	Consumidor por operación de lectura	Un microservicio es consumidor de una entidad E cuando su operación de negocio únicamente lee registros de E para enriquecer su respuesta, sin modificar el estado de E. Los consumidores de E reciben réplicas de E según las reglas del Grupo III.
R-DB-5	Entidad sin propietario claro	Cuando ningún candidato realiza escrituras directas sobre una entidad E (E es solo leída por todos los candidatos), la propiedad se asigna al candidato cuyo conjunto de entidades incluye a E en menor proporción respecto a otras entidades, o al candidato que la referencia con mayor frecuencia. Esta situación es indicativa de una entidad de catálogo o de referencia.
Grupo III		
Reglas de replicación y consistencia eventual		
R-DB-6	Duplicación completa de entidad compartida	Toda entidad compartida E debe replicarse íntegramente con todos sus campos y sus restricciones de integridad pertinentes en la base de datos de cada microservicio consumidor. La réplica es una copia estructural completa, no una vista parcial, para garantizar la autonomía operacional del consumidor sin dependencias de consulta hacia el propietario.
R-DB-7	Independencia del esquema	El esquema de datos de cada microservicio tanto el de las entidades de su propiedad como el de las réplicas de entidades consumidas debe ser desplegable de forma independiente, sin claves foráneas inter-esquema hacia bases de datos de otros microservicios. Las relaciones entre entidades de distintos microservicios se representan únicamente mediante identificadores (ID's) que son identificados localmente dentro de cada microservicio.
R-DB-8	Notificación de cambio por el propietario	Cuando el microservicio propietario de una entidad E ejecuta una operación que crea, modifica o elimina un registro de E, debe emitir una notificación de cambio accesible a todos los consumidores de E. El mecanismo de notificación puede ser

Regla	Nombre	Descripción formal
		sincrónico (patrón Observador) en memoria, para implementaciones de pruebas.
R-BD-9	Actualización de réplica por el consumidor	Al recibir una notificación de cambio sobre una entidad E de la que es consumidor, el microservicio debe actualizar su réplica local de E en su propia base de datos: persistir el nuevo estado del registro si la notificación es de creación o actualización, o eliminar el registro de la réplica local si la notificación es de eliminación.

Modelo de Consistencia Eventual Resultante

La aplicación conjunta de las reglas R-BD-6 a R-BD-9 produce un modelo de consistencia eventual entre las réplicas distribuidas de cada entidad compartida. En este modelo, el estado de una réplica en un microservicio consumidor puede estar temporalmente desactualizado respecto al estado del registro en el microservicio propietario, durante el intervalo de tiempo entre la emisión de la notificación de cambio y su procesamiento por el consumidor.

Esta característica “la consistencia eventual en lugar de la consistencia fuerte de un sistema centralizado” es una consecuencia directa del principio de base de datos por microservicio y es aceptada en la arquitectura de microservicios como costo (trade-off) necesario para obtener las propiedades de autonomía, escalabilidad independiente y resiliencia.

El diagrama conceptual de la Tabla 12 describe el modelo resultante para el caso de una entidad E compartida entre un microservicio propietario MP y dos microservicios consumidores MC₁ y MC₂: MP posee la tabla E en su base de datos y emite notificaciones de cambio; MC₁ y MC₂ poseen réplicas locales de E en sus propias bases de datos y las actualizan al recibir las notificaciones.

Tabla 12 Modelo de separación de datos para una entidad compartida E.

MP (Propietario de E)	MC ₁ (Consumidor de E)	MC ₂ (Consumidor de E)
Base de datos propia: Tabla E (original) <i>Otras tablas propias de MP</i> → Emite: CambioE(crear/actualizar/eliminar)	Base de datos propia: Tabla E (réplica) <i>Tablas propias de MC₁</i> ← Recibe y persiste CambioE	Base de datos propia: Tabla E (réplica) <i>Tablas propias de MC₂</i> ← Recibe y persiste CambioE

Caso Especial: Entidades de Referencia o Catálogo

Un caso particular de entidad compartida es la denominada entidad de referencia o de catálogo, que es accedida en modo solo-lectura por la mayoría de los microservicios candidatos y modificada raramente o solo por algunos procesos. Ejemplos típicos son catálogos de productos, tablas de países o ciudades, listas de tipos o categorías, y tablas de configuración del sistema.

Para este tipo de entidades, el método aplica las mismas reglas de replicación (R-BD-6 a R-BD-9), pero reconoce que la frecuencia de notificaciones de cambio será baja y que el impacto de la consistencia eventual es mínimo para los consumidores. El método sugiere adicionalmente que

estas entidades sean candidatas a un microservicio de catálogo dedicado (Catalog Service o Reference Data Service), que actúa como único propietario y fuente de verdad para todos los consumidores.

3.8 Fase 6: Generación de Artefactos del Microservicio

La sexta y última fase del método transforma la información acumulada en las fases anteriores como son: el conjunto de candidatos con sus entidades, dependencias y marcadores de compartición en artefactos de código fuente y configuración que constituyen la estructura inicial de cada microservicio. Los artefactos generados son completos en estructura, pero con código de negocio marcado para revisión manual, siguiendo el principio de que la herramienta automatiza la extracción y plantillas iniciales (scaffolding), pero la responsabilidad de la corrección del comportamiento de negocio recae en el equipo de desarrollo.

Los artefactos generados para cada microservicio candidato son los siguientes, independientemente del lenguaje o framework de la implementación concreta:

1. Estructura de directorios del proyecto: organizada según las convenciones del sistema de construcción del lenguaje empleado (Maven para Java, MSBuild para C#, pip/poetry para Python, npm/yarn para Node.js).
2. Descriptor de dependencias del proyecto: que declara las dependencias del framework web, el ORM o mecanismo de persistencia, la biblioteca de documentación de API (equivalente a Swagger/OpenAPI), y la base de datos embebida o de prueba.
3. Archivo de configuración del microservicio: que especifica el puerto HTTP asignado, el nombre del servicio, la URL de conexión a la base de datos local, y las propiedades de gestión automática del esquema.
4. Clases de las entidades del dominio: una por cada entidad en el conjunto de entidades del candidato, con sus campos, tipos de datos y anotaciones de mapeo a la base de datos relacional local. Las entidades replicadas (consumidas) son estructuralmente idénticas a las originales del monolito, pero pertenecen al paquete o espacio de nombres del microservicio correspondiente.
5. Interfaces y clases de componentes de acceso a datos: una por cada entidad del candidato, exponiendo las operaciones CRUD estándar más las consultas específicas utilizadas por la operación de negocio.
6. Componentes de lógica de negocio y de entrada: derivados de los componentes equivalentes del monolito, con los “*imports*” actualizados al nuevo espacio de nombres del microservicio y con comentarios “TODO” que señalan los puntos donde la lógica de negocio requiere adaptación al contexto del microservicio.

7. Script de inicialización del esquema de datos: instrucciones DDL (CREATE TABLE) para crear todas las tablas correspondientes a las entidades del candidato en la base de datos local del microservicio.
8. Código del patrón Observador (cuando aplica): para cada entidad compartida del candidato, se generan los cuatro componentes del patrón de notificación de cambios definidos en las Reglas R-BD-8 y R-BD-9.

3.9 Enfoque general del método.

El método se basa en un análisis léxico-sintáctico del código fuente de algún monolito de entrada (Software Legado en Operación) que es un activo de una empresa. Para la correcta migración de un sistema monolítico hacia microservicios se han identificado los siguientes puntos:

1. Identificar los límites de cada microservicio: Identificar todas las clases y sus interacciones mediante llamadas a métodos, funciones y referencias que realicen una meta de valor utilizando el enfoque que considera a un candidato a microservicio como el resultado de un caso de uso o requerimiento del sistema monolítico. Es importante mencionar que la granularidad del microservicio puede incluir una secuencia de varias funciones que todas en conjunto pueden convertirse en un microservicio.
2. Obtener lista de archivos de los candidatos a microservicios y los archivos de las bases de datos independientes correspondientes a cada microservicio: Generar bases de datos separadas, basadas en las relaciones de datos y las interacciones de las clases y sus dependencias.
3. Dependencias de datos: Determinar qué modelos de datos son manipulados por N clases de diferentes casos de uso y de esta forma identificar si tienen relación con otros candidatos a microservicios que ocupen el mismo modelo de datos (tablas accedidas en diferentes procesos del sistema).
4. Implementación de la API y del patrón observador o de envío de mensajes para comunicar los microservicios.
5. Despliegue de los microservicios.
6. Implementar las pruebas de validación.

A continuación, se describen los pasos del método para descomponer la base de datos de sistemas monolíticos en base al proceso BPMN presentado en la figura 3.

3.9.1 Análisis del sistema monolítico

- Preparación de Archivos: Colocar los archivos Java en un directorio específico para el análisis (proyecto-paquete) del sistema monolítico legado.
- Verificar que el sistema legado esté libre de errores y se encuentre desarrollado en lenguaje Java.

Consideraciones adicionales:

- Asegurar que el Visitor detecte invocaciones a métodos y referencias a otras clases.
- Implementar una estructura de almacenamiento temporal para registrar las interacciones y dependencias de datos, con el objetivo de medir el acoplamiento entre servicios.

3.9.2. Identificación de microservicios

- Configuración de ANTLR: Implementar gramáticas completas de Java y SQL que permitan el reconocimiento de clases, métodos, atributos y consultas SQL complejas.
- Desarrollo del Patrón de diseño Visitor: Crear un patrón de diseño Visitor personalizado en el lenguaje seleccionado por ejemplo Java que recorra el árbol de sintaxis y capture las relaciones entre las entidades (clases) y sus interacciones con los métodos y funciones, incluyendo el modelo de datos.
- Identificación de secuencias de interacción: Detectar secuencias de métodos que interactúan con el modelo de datos (p. ej., creación, lectura, actualización y eliminación de entidades).
- Mapeo de Clases y Métodos: Utilizar el patrón Visitor para construir un mapa/listas que relacione cada clase con los métodos y funciones que contiene y las llamadas a otras clases.
- Referencias Cruzadas: Comprobar referencias entre clases para identificar relaciones de dependencia.
- División de la Base de Datos: Basado en el mapeo y las secuencias de interacción, identificar el modelo de datos manejados por cada posible microservicio.
- Creación de Esquemas: Generar automáticamente los esquemas y la base de datos para cada microservicio en lenguaje SQL.
- Integridad Referencial: Asegurar que las relaciones entre tablas del monolito original se mantengan en cada base de datos separada.

3.9.3 Generación de Gramática y SQL

- Lenguaje de programación:
 - Diseñar una gramática que permita identificar las clases y métodos relevantes para la interacción con la base de datos.
 - Analizar patrones de acceso a datos en el código, como EntityManager o Session en JPA, para mapear las entidades asociadas a los casos de uso.

- SQL:
 - Crear una gramática que reconozca las estructuras de las tablas, relaciones, restricciones, y dependencias en los scripts SQL existentes.
 - Establecer un modelo de mapeo entre las entidades Java y las tablas SQL para entender la relación entre código y datos.

3.9.4 Configuración del Analizador Léxico-Sintáctico

- Utilizar **ANTLR** para generar analizadores basados en las gramáticas del lenguaje seleccionado y SQL.
- Configurar el analizador para:
 - Extraer árboles sintácticos que representen las relaciones entre clases, métodos y demás elementos implicados en cada secuencia interactiva del caso de uso correspondiente.
 - Identificar transacciones y dependencias entre tablas desde el código fuente y los scripts SQL.

3.9.5 Implementación del Patrón de Diseño Visitor

- El patrón Visitor debe navegar los árboles sintácticos generados por el analizador para:
 - Detectar las entidades asociadas a cada microservicio.
 - Identificar las dependencias directas entre tablas o datos compartidos entre microservicios.
- Diseñar métodos en el Visitor para:
 - Agrupar entidades relacionadas dentro de un mismo contexto delimitado o caso de uso.
 - Generar esquemas SQL específicos para cada microservicio, respetando las relaciones internas.

3.9.6 Generación Automática de Código para Microservicios

- Esqueleto de Microservicio: Crear el código para cada microservicio que incluya la lógica de negocio necesaria y las interfaces para acceder a su base de datos.
- Esqueleto de Scripts SQL de Base de Datos: Generar scripts que creen la estructura de la base de datos para cada microservicio.
- Configuración de Conectores: Asegurar que cada microservicio tenga la configuración necesaria para conectarse a su base de datos específica.

3.10 Estructura de almacenamiento de datos

Para detallar cómo se almacena la información de las estructuras de datos de los candidatos a microservicios, se presenta un ejemplo basado en un caso de uso. En el diseño de sistemas de software, un caso de uso representa una interacción específica entre un actor (que puede ser un usuario o un sistema externo) y el sistema, con el propósito de alcanzar un objetivo definido. Los elementos que componen un caso de uso son fundamentales, ya que describen y definen sus características esenciales.

3.10.1 Elementos de un caso de uso.

1. Clase Iniciadora (Entry Point)

- Es la clase pública que inicia la lógica del caso de uso. Normalmente contiene un método público que sirve como punto de entrada.
- Ejemplo: un controlador en un patrón MVC o una clase con un método *main*.

2. Métodos

- Métodos definidos en la clase iniciadora o en clases relacionadas.
- Pueden ser:
 - Llamadas internas: Llaman a métodos dentro de la misma clase.
 - Llamadas externas: Llaman a métodos de otras clases.
 - Métodos con operaciones sobre objetos, cálculos, transformaciones.
 - Calificadores (Abstract, Public, Private ...)

3. Atributos

- Variables definidas en la clase. Pueden ser:
 - Primitivas (instancia y clase):
 - Variables primitivas de instancia:
 - Calificadores de alcance (private, protected, friendly, public)
 - Calificadores de tipo (int, float, boolean, etc).
 - Variables primitivas de clase:
 - Calificadores de alcance (private, protected, friendly, public)
 - Calificadores de tipo (int, float, boolean, etc).
- Relaciones entre Clases
 - Dependencias entre clases, como:
 - Asociaciones directas (composición, agregación y dependencia).
 - Herencia (extends).
 - Implementaciones de interfaces (implements).

4. Interfaces

- Definiciones abstractas implementadas por clases. Representan contratos que las interfaces deben cumplir.

5. Objetos Anidados

- Clases internas o anónimas definidas dentro de una clase.

6. Propiedades de Configuración

- Constantes (static final).
- Valores cargados desde archivos de configuración o bases de datos.

7. Manejo de Excepciones

- Bloques try-catch para manejar errores.

8. Dependencias de Paquetes

- Importaciones (import) que indican relaciones con otros paquetes o bibliotecas.

En la figura 4, se muestran las principales relaciones de interacción entre los elementos de un caso de uso, que es considerado como un posible microservicio.

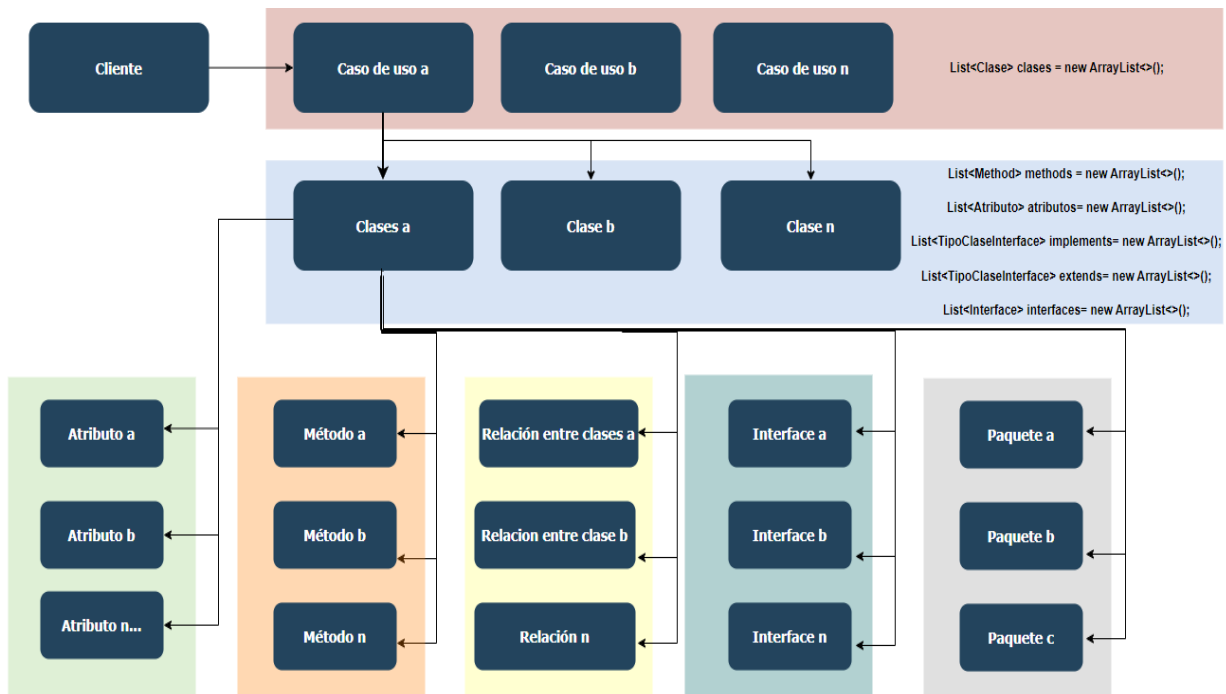


Figura 4 Mapa de las relaciones entre los elementos del caso de uso.

3.10.2 Elementos Específicos de Base de Datos (SQL)

1. Consultas SQL

- Consultas de lectura:
 - SELECT para extraer datos.
- Consultas de modificación:
 - INSERT, UPDATE, DELETE.
- Ejecuciones directas o mediante frameworks como JDBC, JPA-Java Persistence API, Hibernate, etc.
-

2. Tablas y Columnas Involucradas

- Identificar:
 - Tablas leídas.
 - Columnas seleccionadas, modificadas o filtradas.

3. Operaciones transaccionales

- Operaciones agrupadas en una transacción (BEGIN, COMMIT, ROLLBACK).

4. Uso de Procedimientos Almacenados

- Llamadas a funciones o procedimientos almacenados.

5. Parámetros Dinámicos

- Parámetros que se pasan a las consultas (en JDBC, nombrados parámetros en JPA-Java Persistence API).

6. Conexiones y Configuración

- Strings de conexión.
- Pools de conexiones.

En la figura 5, se muestran las relaciones de interacción entre los elementos de un método, que es parte de la secuencia de interacción del caso de uso que atiende una meta de valor.

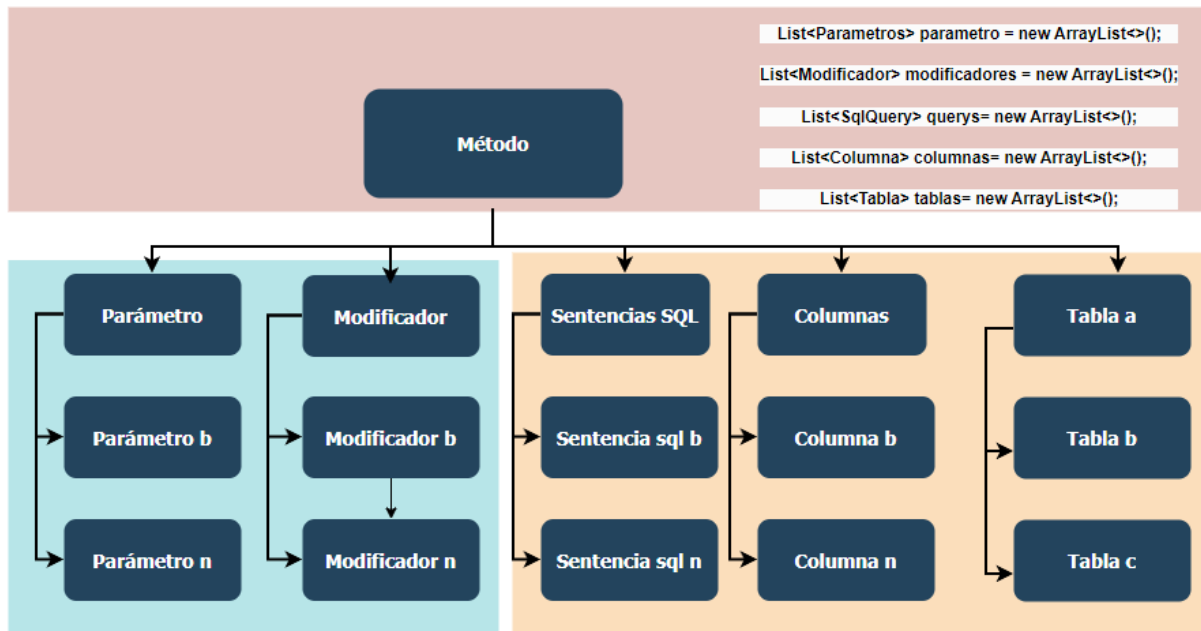


Figura 5 Mapa de las relaciones de un método perteneciente a un caso de uso.

3.11 Implementación del Método en la Herramienta Informática

La herramienta informática que automatiza el método propuesto se denomina TOOLMONO2MSDB, es una aplicación web de escritorio ejecutada localmente, construida con Java 17 y Spring Boot 3.2.5, que integra un motor de análisis estático basado en ANTLR4, un generador de artefactos Maven y una interfaz de usuario web implementada con Thymeleaf y Bootstrap 5. Opera completamente en la máquina local del usuario sin dependencias de servicios externos.

El ciclo de vida de la implementación comprende cinco etapas: (1) selección del directorio fuente del monolito, (2) análisis y visualización de candidatos a microservicio, (3) inspección del detalle de cada candidato, (4) generación de artefactos Maven, y (5) despliegue y verificación de los microservicios vía Swagger UI.

Arquitectura en Módulos Maven

La herramienta se organiza en dos módulos Maven con responsabilidades diferenciadas como se puede observar en la tabla 13:

Tabla 13 Módulos Maven de la herramienta MONO2MSDB.

Módulo	Artefacto	Responsabilidad
extractor-core	extractor-core-1.0.0.jar	Motor de análisis estático (ANTLR4), generación de artefactos, cálculo de métricas IATC y modelos de datos. Sin dependencias con Spring Boot, reutilizable como biblioteca independiente.
extractor-ui	extractor-ui-1.0.0.jar	Aplicación Spring Boot MVC con interfaz web Thymeleaf. Depende del extractor-core. Expone los endpoints HTTP y gestiona el ciclo de vida de la sesión de análisis. Servidor Tomcat embebido en puerto 9090.

A continuación, la tabla 14 muestra las dependencias tecnológicas de la herramienta.

Tabla 14 Dependencias tecnológicas de la herramienta.

Dependencia	Versión	Módulo	Función
Spring Boot Starter Web	3.x	extractor-ui	Servidor Tomcat embebido, dispatcher MVC, serialización JSON.
Thymeleaf + Layout Dialect	3.x / 3.2.1	extractor-ui	Motor de plantillas HTML del lado del servidor con soporte de plantilla maestra.
ANTLR4 Runtime	4.13.x	extractor-core	Análisis léxico-sintáctico del código Java del monolito para construcción del AST.
Jackson Databind	2.17.1	ambos	Serialización de candidatos a JSON y respuestas REST.
Bootstrap 5 + Chart.js	CDN	extractor-ui	Framework CSS responsivo y gráfica de barras comparativa del IATC.

3.11.1 Diagrama de Casos de Uso

La Figura 6 presenta el diagrama general de casos de uso de la herramienta generado con el lenguaje PlantUML. El cliente es el único actor externo del sistema e interactúa con siete casos de uso interrelacionados mediante las relaciones «include» y «extend» del estándar UML 2.5. Las relaciones «include» modelan la parte obligada de ejecución de un caso de uso base antes de otro; la relación «extend» modela la ejecución condicional de “Analizar monolito” solo cuando el directorio seleccionado es válido.

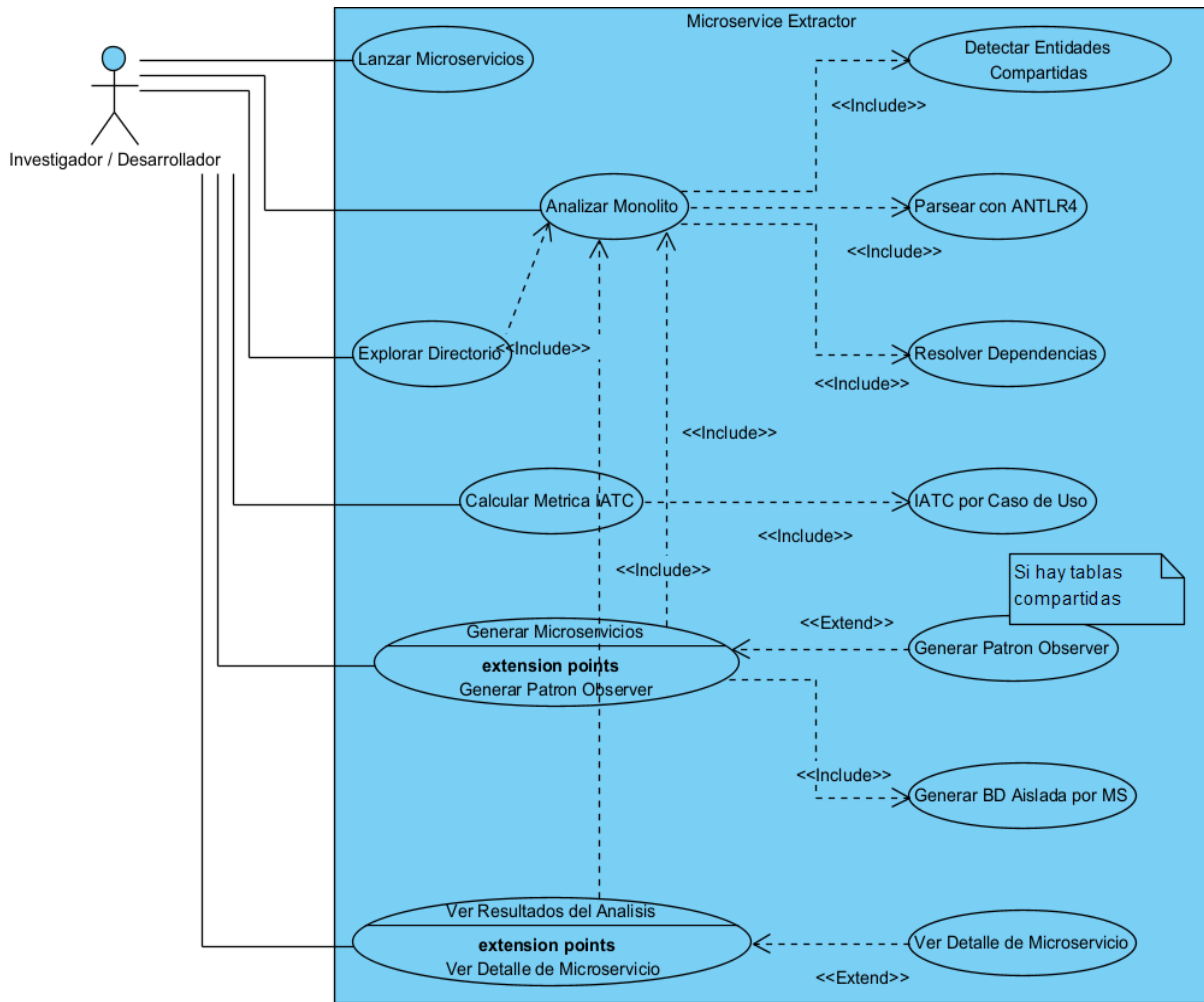


Figura 6 Diagrama general de casos de uso de la herramienta MONO2MSDB.

Tabla 15 Descripción general de los casos de uso que intervienen en la herramienta.

ID	Caso de Uso	Actor	Descripción
UC1	Analizar Monolito	Cliente	Recibe la ruta del proyecto monolito en Java, aplica análisis léxico-sintáctico con ANTLR4 y el patrón Visitor para identificar candidatos a microservicios, sus entidades JPA y dependencias entre clases. Es el caso de uso central de la herramienta.
UC2	Explorar Directorio	Cliente	Abre un selector de carpeta nativo del sistema operativo para que el usuario navegue y elija la ruta del monolito sin necesidad de escribirla manualmente.
UC3	Ver Resultados del Análisis	Cliente	Presenta la lista completa de candidatos identificados: nombre del microservicio, caso de uso detectado, puerto asignado, entidades propias y entidades compartidas con otros candidatos. Requiere que UC1 haya sido ejecutado previamente.
UC4	Ver Detalle de Microservicio	Cliente	Muestra el detalle de un candidato específico: clases de servicio asociadas, repositorios, entidades JPA, dependencias

ID	Caso de Uso	Actor	Descripción
			y si el candidato requiere generación del patrón Observer. Extiende a UC3.
UC5	Generar Microservicios	Cliente	Genera la estructura completa de proyectos Spring Boot independientes para cada candidato identificado, incluyendo pom.xml, application.properties con base de datos propia, entidades, repositorios, servicios y controladores REST. Requiere que UC1 haya sido ejecutado previamente.
UC6	Calcular Métrica IATC	Cliente	Calcula la métrica IATC (Inter-service Access to Table Coupling) para el monolito y para cada microservicio generado, presentando el porcentaje de reducción de acoplamiento a nivel de acceso a tablas conseguido con la extracción.
UC7	Lanzar Microservicios	Cliente	Ejecuta en paralelo todos los microservicios generados mediante mvn spring-boot:run, arrancando cada uno en su puerto asignado (8081, 8082, 8083...). Requiere que UC5 haya sido ejecutado previamente.
UC8	Parsear con ANTLR4	<<include>> de UC1	Por cada archivo .java del monolito aplica JavaLexer para tokenizar el código fuente, CommonTokenStream para gestionar el flujo de tokens y JavaParser.compilationUnit() para construir el árbol sintáctico abstracto (AST) sobre el que opera el Visitor.
UC9	Resolver Dependencias Cross-file	<<include>> de UC1	Una vez procesados todos los archivos, cruza los mapas acumulados por el Visitor (serviceToRepositories, repositoryToEntity, entityFields) para conectar cada candidato con sus repositorios y, a través de las relaciones @OneToMany / @ManyToOne, con sus entidades JPA transitivas.
UC10	Detectar Entidades Compartidas	<<include>> de UC1	Consulta el mapa entityToCandidates del resultado: si una entidad @Entity aparece referenciada por más de un candidato se marca como compartida y cada candidato afectado recibe hasSharedEntities = true. Esta es la decisión algorítmica que define la separación del modelo de datos y activa la generación del patrón Observer.
UC11	Generar BD Aislada por Microservicio	<<include>> de UC5	Genera el archivo application.properties de cada microservicio con spring.datasource.url=jdbc:h2:mem:{useCaseName}db, asignando a cada microservicio su propia base de datos H2 en memoria completamente aislada. Materializa el patrón Database per Service.
UC12	Generar Patrón Observer	<<extend>> de UC5, condición: hasSharedEntities = true	Se activa únicamente cuando dos o más microservicios comparten una entidad JPA. Genera automáticamente los componentes de sincronización eventual: XxxEntityListener (@PostPersist), XxxChangePublisher (notificación REST post-commit), ObserverNotificationController (receptor en el microservicio par), XxxChangeObserver (persistencia de la copia local), ObserverContext (bandera anti-ciclo ThreadLocal) y ApplicationContextHolder.
UC13	IATC por Caso de Uso	<<include>> de UC6	Calcula la métrica IATC individual para cada caso de uso identificado.

El diagrama de caso de uso de la Figura 7 presenta el caso de uso principal con las interacciones entre el actor principal “Cliente” y las funcionalidades que expone la herramienta. El proceso inicia con la migración de los candidatos a microservicios identificados el cual incluye la separación del modelo de datos, y extiende a otro caso de uso para mantener la consistencia de la información cuando haya tablas compartidas entre microservicios, permitiendo al cliente evaluar el acoplamiento e inspeccionar el detalle de cada candidato individualmente y generar los artefactos Maven (microservicios y su base de datos) listos para ser desplegados.

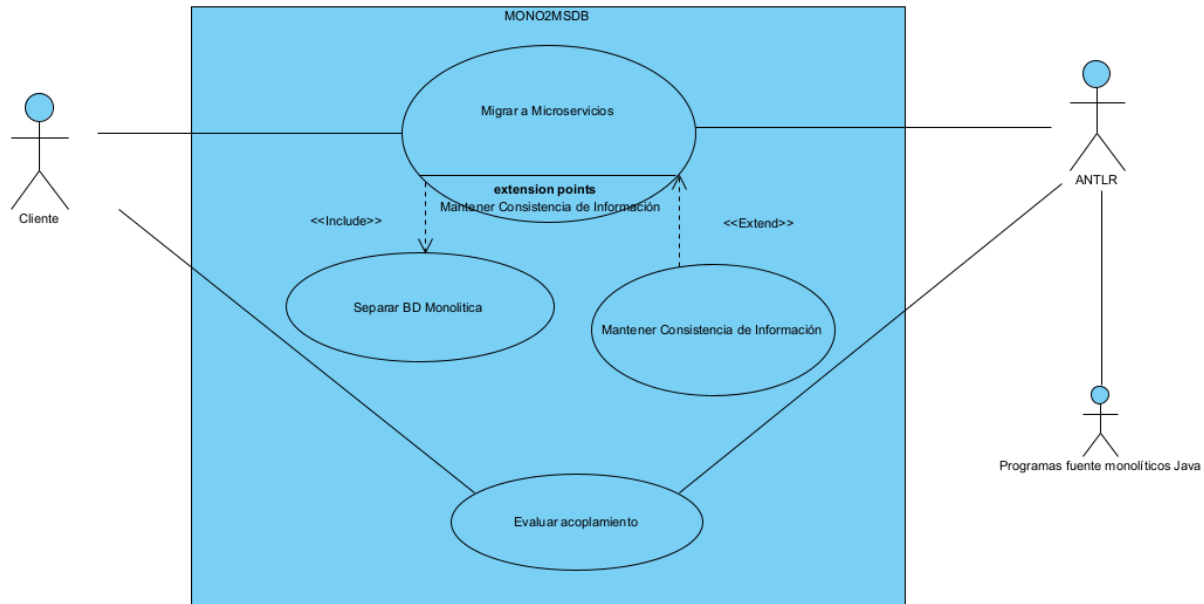


Figura 7 Diagrama principal de caso de uso.

3.11.2 Diagrama de secuencias

El diagrama de secuencias de la Figura 8 describe el flujo de mensajes entre los componentes principales de la herramienta durante la ejecución de las dos operaciones centrales del método: el análisis del monolito y la generación de artefactos. En la fase de análisis, el Cliente ingresa la ruta del proyecto a través de la interfaz web; el ExtractorWebController delega la ejecución a la entidad MicroserviceAnalyzer, el cual invoca al JavaFileVisitor basado en el patrón Visitor de ANTLR4 para recorrer y clasificar cada archivo fuente. Los candidatos detectados se encapsulan en la entidad MonolithAnalysisResult y se devuelven como respuesta en formato JSON para su visualización. En la fase de generación, el controlador delega la acción al MicroserviceGenerator, que produce la estructura Maven completa de cada microservicio candidato incluida su base de datos e informa la ruta de salida al cliente.

En el Bloque 1 (POST /analyze), el Controller orquesta el análisis completo: instancia el Analyzer, que crea el visitante (Visitor) y lo aplica sobre todos los archivos .java del monolito; los métodos de visita acumulan componentes y candidatos en MonolithAnalysisResult; al terminar el ciclo, el Analyzer ejecuta las fases de resolución y detección secuencialmente y

retorna el resultado. En el Bloque 2 (POST /generate), el Generator itera sobre los candidatos produciendo todos sus artefactos, con generación condicional del patrón Observador. En el Bloque 3 (GET /coupling), el Calculator aplica la fórmula IATC dos veces y retorna las métricas comparativas.

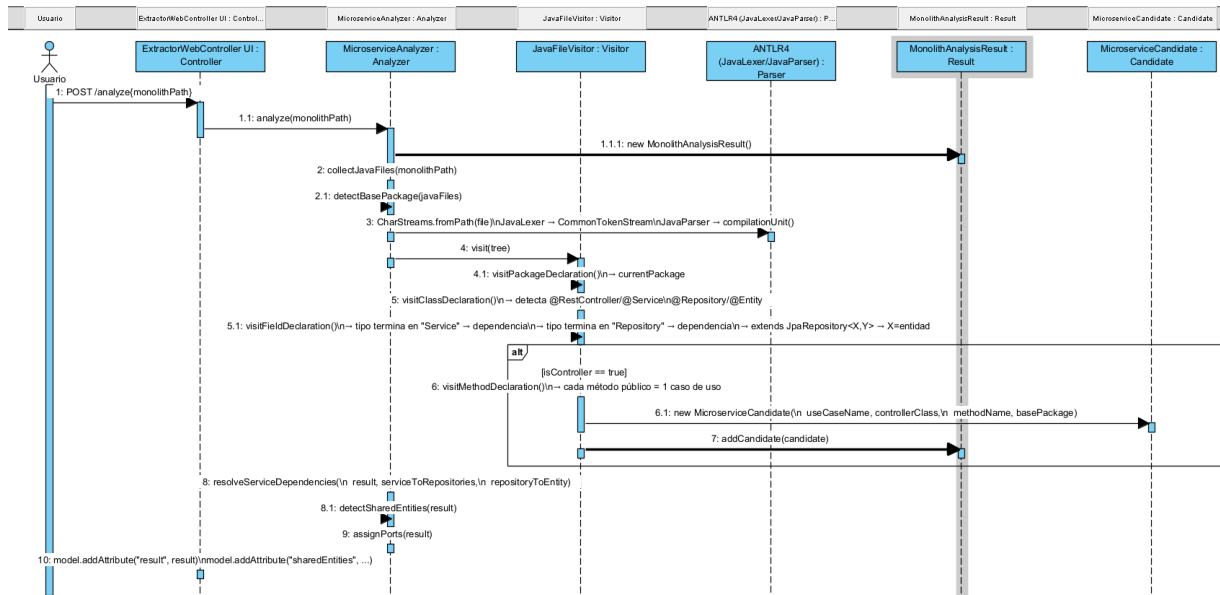


Figura 8 Diagrama de secuencia de la herramienta MONO2MSDB.

3.11.3 Diagrama de clases de la implementación de la herramienta.

El diagrama de clases de la Figura 9 presenta la arquitectura interna de la herramienta a través de sus ocho clases principales, organizadas en dos capas. La capa de control agrupa las clases responsables del procesamiento: la clase ExtractorWebController actúa como punto de entrada HTTP y coordina las operaciones; MicroserviceAnalyzer implementa las seis fases del método de extracción; JavaFileVisitor aplica el patrón Visitor sobre el árbol sintáctico generado por ANTLR4; MicroserviceGenerator produce los artefactos Maven de cada microservicio candidato; y CouplingCalculator implementa la métrica IATC comparativa entre el monolito y la arquitectura propuesta. La capa de modelo concentra las estructuras de datos: MonolithAnalysisResult agrega mediante composición los candidatos identificados (MicroserviceCandidate), y CouplingMetrics encapsula los valores numéricos del índice de acoplamiento. Las dependencias de uso entre capas reflejan el principio de separación de responsabilidades que guía el diseño de la herramienta.

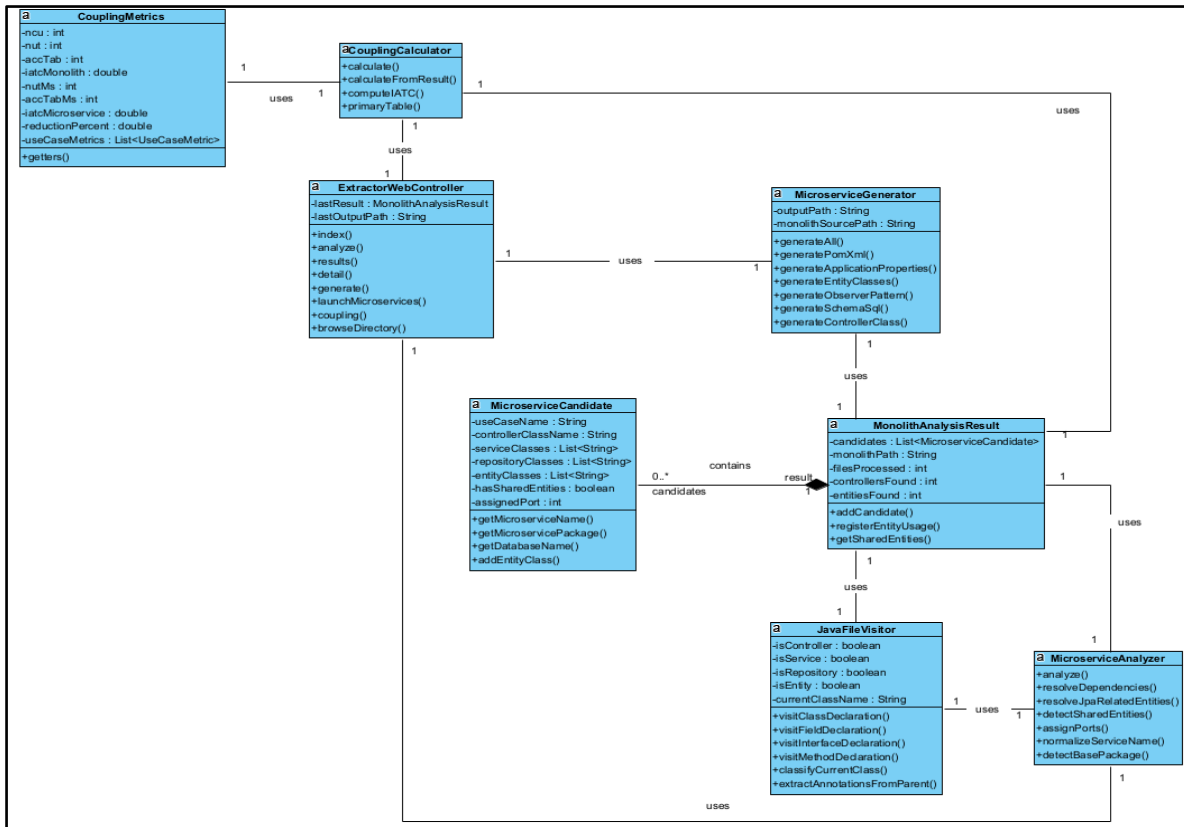


Figura 9 Diagrama de clases de la herramienta informática.

Capítulo 4: Pruebas

4.1 Evaluación del método propuesto

Esta sección presenta la estrategia de evaluación utilizada para validar el método de descomposición de bases de datos monolíticas hacia una arquitectura de microservicios bajo el enfoque de una base de datos por servicio.

La evaluación se realizó en dos etapas:

1. Validación de la herramienta que implementa y automatiza el método.
2. Evaluación experimental de la reducción del acoplamiento indirecto.

4.2. Validación del método en herramienta informática

En este apartado, se describe el plan de pruebas utilizado para evaluar el método de descomposición de sistemas monolíticos aplicado en una herramienta informática, con un enfoque en la descomposición de la base de datos. El proceso de pruebas sigue el estándar ISO/IEC/IEEE 29119, que establece un marco detallado para el ciclo de vida de las pruebas de

software. A través de este enfoque estructurado, se busca demostrar que el método de descomposición cumple con los objetivos propuestos, logrando mediante su aplicación, disminuir el acoplamiento y mejorando la autonomía de los microservicios resultantes.

4.2.1 Alcance

Este documento está dividido en secciones que siguen las etapas definidas en el estándar ISO/IEC/IEEE 29119: Plan de Pruebas, Diseño de Pruebas, Ejecución de Pruebas y Evaluación de Resultados. A lo largo de este apartado, se incluyen ejemplos completos de pruebas en cada etapa, junto con los formatos de documentación requeridos por el estándar. Esto permite demostrar cómo cada fase del proceso de pruebas contribuye a la validación de los resultados obtenidos de la aplicación del método a la herramienta de descomposición y su efectividad para descomponer la base de datos de un sistema monolítico en una arquitectura de microservicios.

Referencias utilizadas en este documento

- Reporte de análisis del método de descomposición.
- Reporte de diseño del método de descomposición (diagrama de procesos BPMN).
- Documento de diseño de la herramienta implementada con el método de descomposición de la base de datos.
- Estándar ISO/IEC/IEEE 29119

Glosario e identificadores del documento

- UAT: Pruebas de Aceptación del Usuario (del inglés User Acceptance Test).
- TOOLMONO2MSDB = Herramienta de descomposición de la base de datos monolítica hacia microservicios.

La convención para los identificadores del documento de pruebas es la siguiente:

- TOOLMONO2MSDB = Herramienta de descomposición.

Etapas del plan de pruebas:

- NP = Nivel de Pruebas.
- PP = Plan de Pruebas.
- DP = Diseño de Pruebas.
- CP = Casos de Pruebas.
- EP = Ejecución de Pruebas.
- ER = Evaluación de Resultados.

Identificador:

- XX = Identificador numérico incremental.

Ejemplo: **TOOLMONO2MSDBPP01** – Documento de plan de pruebas

4.2.2 Plan de pruebas

Documento de plan de prueba - **ISTOOLMONO2MSDBPP01**
Instrumento de Pruebas

- **Herramienta informática - TOOLMONO2MSDB:** Implementa y automatiza el método para descomponer la base de datos de sistemas monolíticos para operar en arquitectura de Microservicios.
- **Versión:** 1.0
- **Tecnología usada:** Java 17, ANTLR4 versión 4.13.1 (gramática java), Lenguaje SQL.
- **IDE de Pruebas:** Eclipse Versión 4.37.
- **Base de datos:** Base de datos en memoria (H2-Database)

Objetivos del plan de pruebas

- Verificar la generación de bases de datos por microservicio.
- Evaluar la equivalencia funcional de los microservicios con el monolítico original.

Estrategia y elementos de pruebas

La estrategia de prueba se estructura en dos niveles de prueba progresivos, conforme a la taxonomía del estándar ISO/IEC/IEEE 29119-2, aplicados de manera independiente sobre cada monolito de prueba y microservicios generados en, al menos, 3 casos de uso por sistema como se puede observar en la tabla 16.

Tabla 16 Niveles y tipos de pruebas

Nivel	Tipo de Prueba	Descripción	Técnica (ISO 29119-4)
NP-1	Validación por observación	Verificar la estructura de la base de datos generada por microservicio: existencia de esquema, tablas y consistencia con el modelo de datos del monolito original.	Inspección / Revisión estructurada
NP-2	Integración por observación y pruebas unitarias	Evaluar el funcionamiento correcto entre cada microservicio y su base de datos autónoma, comparando el comportamiento funcional con el sistema legado original, mediante pruebas unitarias de capa de servicio.	Prueba de integración / Prueba unitaria basada en especificación

Las pruebas tipo caja negra (o black box testing) son técnicas de prueba de software en las que se evalúa la funcionalidad de una aplicación sin conocer su estructura interna, código fuente o lógica de implementación. Se enfocan únicamente en las entradas y salidas del sistema, no en cómo se procesan. Estas pruebas son válidas y aceptadas por la comunidad de ingeniería de software.

Características que no serán probadas

A continuación, se describen las características que no serán probadas:

- No se incluyen la totalidad de combinaciones sintácticas del lenguaje java.
- Que el código legado esté libre de errores y/o defectos.
- La interfaz al usuario (FrontEnd) no es probada.

Verificación de la correcta generación de las bases de datos para cada microservicio

El criterio de aprobación o desaprobación del proceso para verificar la correcta generación de las bases de datos será mediante observación al generarse el archivo de salida que contendrá el código del microservicio por caso de uso identificado, con su correspondiente base de datos incluidas las tablas identificadas durante el proceso de descomposición.

Criterios de aceptación:

- Cada microservicio cuenta con una base de datos autónoma y funcional.
- La estructura de tablas en cada base de datos corresponde con su parte del modelo de datos del sistema monolítico inicial.

Evaluación del correcto funcionamiento y la consistencia de la información entre los microservicios y sus bases de datos

El criterio de aprobación o desaprobación para evaluar el correcto funcionamiento será comparando el comportamiento de la arquitectura de software antes y después del método de descomposición mediante pruebas unitarias del sistema monolítico, y pruebas de ejecución del sistema de microservicios.

Criterios de aceptación:

- Las pruebas unitarias aplicadas al sistema monolítico se deben ejecutar sin errores.
- Los microservicios funcionan correctamente de forma individual, según los casos de uso definidos.
- Las pruebas unitarias, funcionales y de integración pasan con éxito (por ejemplo, pruebas automatizadas mediante el cliente Postman).

Evaluación de la reducción del acoplamiento entre los servicios

El criterio de aprobación o desaprobación de las pruebas de calidad será mediante la comparación de los resultados obtenidos al aplicar el cálculo de la métrica IATC manualmente, antes y después de la aplicación del método de descomposición.

Criterios de aceptación:

Se aplica la métrica formal de acoplamiento IATC.

- Comparación cuantitativa entre el nivel de acoplamiento en el sistema monolítico original y en los microservicios descompuestos.
- Una disminución comprobada (medible) en porcentaje sobre el acoplamiento entre los microservicios generados en comparación al monolito de origen.

4.2.3 Selección de los casos de prueba

Para la selección de los casos de prueba, se desarrolló inicialmente un sistema monolítico como caso sintético y, adicionalmente, se seleccionaron dos sistemas monolíticos obtenidos en internet, a los cuales se aplicaron las pruebas unitarias y los diseños de las pruebas. En la tabla 17 se describen los componentes del entorno de prueba de la herramienta.

Entorno de las pruebas

Tabla 17 Componente del entorno de pruebas

Componente	Versión / Detalle
Java	OpenJDK 17
Spring Boot	3.2.5
ANTLR	4.13.1
Framework de pruebas	JUnit 5 + Mockito
Base de datos (pruebas)	H2 In-Memory
Herramienta de extracción	microservice-extractor v1.0.0
Puerto UI	localhost:9090
Puertos microservicios	8081 – 8090 (secuencial por candidato)
Swagger UI	http://localhost:{puerto}/swagger-ui.html

4.2.4 Diseño de pruebas

En este apartado se realizaron los diseños de pruebas “Generación de Base de Datos por Microservicio”, “Identificación del Modelo de Datos del Monolito Original”. La descripción del diseño de pruebas se presenta en las tablas 18, 19.

Tabla 18 Generación de bases de datos por microservicio.

Diseño de la Prueba – Generación de Base de Datos por Microservicio	
Identificador	ISMONO2MSDP01
Característica a ser probada	Correcta generación de base de datos y la estructura de tablas por cada uno de los microservicios generados a partir de la identificación del caso de uso con su correspondiente modelo de datos.
Descripción	Verificar si la base de datos de cada microservicio y su estructura de tablas han sido generadas correctamente.
Criterios de Aprobación	Cada microservicio debe tener su propia base de datos, y esta debe estar correctamente estructurada de acuerdo con las tablas del modelo de datos del sistema legado de origen.
Validación	Por observación, verificar la existencia de la base de datos por microservicio.
Refinamiento	Asegurarse de que las bases de datos sean autónomas con sus correspondientes tablas, sin dependencias entre ellas.
Resultado Esperado	Cada microservicio debe tener una base de datos autónoma correctamente generada con sus respectivas tablas relacionadas con el modelo del sistema legado original.

Tabla 19 Identificar el modelo de datos

Diseño de prueba – Verificar el correcto funcionamiento del monolito y los microservicios generados	
Identificador	ISDMONO2MSDP02
Característica a ser probada	Se evaluará el correcto funcionamiento del sistema monolítico legado y los microservicios que implementan el método de descomposición antes y después de su aplicación respectivamente.
Descripción	- Verificar que el sistema legado en operación funcione de forma correcta, mediante la aplicación de pruebas unitarias en al menos 3 casos de uso. - Verificar que el sistema de microservicios funcione de forma correcta en los mismos casos de uso después de aplicar el método de descomposición.
Criterios de aprobación	La aprobación sobre el funcionamiento del sistema legado monolítico y los microservicios generados será dada por la ejecución automática de las pruebas unitarias en el sistema monolítico y la ejecución de los puntos de acceso de los microservicios con Postman o Swagger. Para que esta comparación sea aprobatoria los resultados deben ser iguales.
Evaluación	Evaluación por observación
Refinamiento	Asegurarse de que los resultados de las pruebas unitarias sean exitosas y que los puntos de acceso de los microservicios se ejecuten de forma correcta.
Resultado Esperado	La funcionalidad en ambos sistemas debe ser la misma, tanto en las pruebas unitarias como en las pruebas de funcionalidad.

4.2.5 Casos de pruebas

Los casos de prueba en esta investigación son descritos en la tabla 20, y se limitan a probar al menos 3 casos de uso por sistema monolítico.

Tabla 20 Casos de prueba

Caso de prueba	Módulo	Dominio	Descripción	Entidades
ISMONO2MSCP01	sample-Sales	Sistema de Ventas	Gestiona el proceso de ventas de productos. Permite registrar clientes, administrar el catálogo de productos, generar órdenes de compra y controlar los artículos incluidos en cada pedido. Este sistema concentra parte de la lógica relacionada con la comercialización de productos y la captura de transacciones de venta.	Categoría, Orden, Producto, Cliente, pedido y detalle del pedido
ISMONO2MSCP02	sample-Billings	Facturación y Devoluciones	Administra los procesos financieros posteriores a una venta. Se encarga de generar facturas, aplicar descuentos comerciales y gestionar devoluciones de productos. Centraliza la lógica relacionada con la emisión de comprobantes, ajustes económicos y control de reembolsos.	Invoice, Return, Discount
ISMONO2MSCP03	sample-Loyalty	Fidelización de Clientes	Gestiona los programas de lealtad y recompensas para los clientes. Permite mantener cuentas de fidelización, registrar acumulación y canje de puntos, así como administrar recompensas disponibles.	LoyaltyAccount, Reward, PointTransaction

Caso de prueba ISMONO2MSCP01 – Sistema de gestión de ventas

El sistema monolítico objeto de prueba integra funcionalidades relacionadas con la gestión de ventas, a través de los controladores CategoriaController, ProductoController, ClienteController, PedidoController y DetallePedidoController. Estas funcionalidades permiten administrar el catálogo de productos, registrar y mantener información de clientes, así como procesar pedidos y sus componentes asociados. A partir de estas operaciones, se generan interacciones entre clases y el modelo de datos, cuyas dependencias se representan en el diagrama de clases como se muestra en la figura 10.

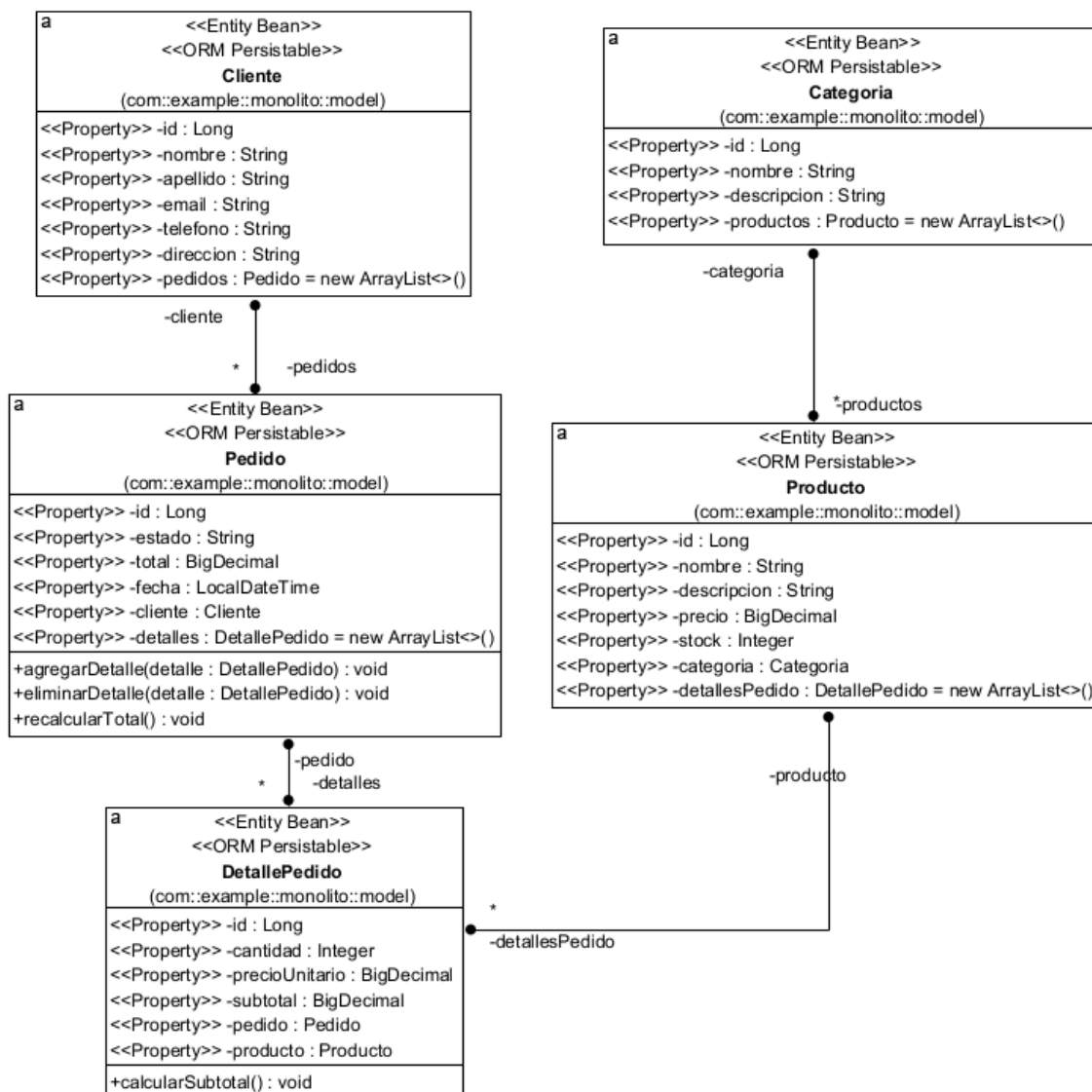


Figura 10 Diagrama de clases del sistema de ventas

En el diagrama de clases se pueden observar las relaciones entre los modelos de datos y con base en el análisis del código fuente se obtuvieron las siguientes relaciones entre los casos de uso (controladores) y el modelo de datos:

- CategoriaController → modelo Categoria
- ClienteController → modelo Cliente
- DetallePedidoController → modelo DetallePedido
- PedidoController → modelo Pedido
- ProductoController → modelo Producto

Estas clases representan capacidades de negocio que interactúan con entidades del modelo de datos (y, por ende, tablas en la base de datos). En el proceso de análisis y revisión del código se obtuvieron las relaciones entre las tablas y el modelo de datos como se muestra en las tablas 21 y 22 donde se describe la relación de casos de uso con el modelo de datos:

Tabla 21 Relaciones entre las tablas del sistema de gestión de ventas.

Modelo	Relacionado con	Relación típica (JPA)
Pedido	Cliente	@ManyToOne
Pedido	DetallePedido	@OneToMany
DetallePedido	Producto	@ManyToOne
Producto	Categoria	@ManyToOne

Tabla 22 Relación de casos de uso con el modelo de datos del sistema de gestión de ventas.

Capacidad de negocio	Categoria	Producto	Pedido	DetallePedido	Cliente
CategoriaController	✓				
ProductoController	✓	✓			
PedidoController			✓	✓	✓
DetallePedidoController		✓	✓	✓	
ClienteController			✓		✓

Procedimiento de Ejecución de la prueba

Los resultados de la ejecución del plan de pruebas realizada en el sistema de gestión de ventas se muestran en la tabla 23.

Tabla 23 Generación de bases de datos por microservicio para el sistema de gestión de ventas.

Nombre de la prueba	ISMONO2MSCP01
Diseño de prueba	ISMONO2MSDP01
Sistema legado (monolito)	Sistema de gestión de ventas
Entrada	Código fuente del sistema legado monolítico en la herramienta que implementa el método.
Objetivo	Verificar que la base de datos de cada microservicio y su estructura de tablas haya sido generada correctamente.
Microservicio Categoría	Cap. de Negocio:

	monito Nombre controller repository service monito > controller Nombre CategoriaController monito > repository Nombre CategoriaRepository monito > service Nombre CategoriaService Base de datos: resources Nombre db application resources > db Nombre schema
--	---

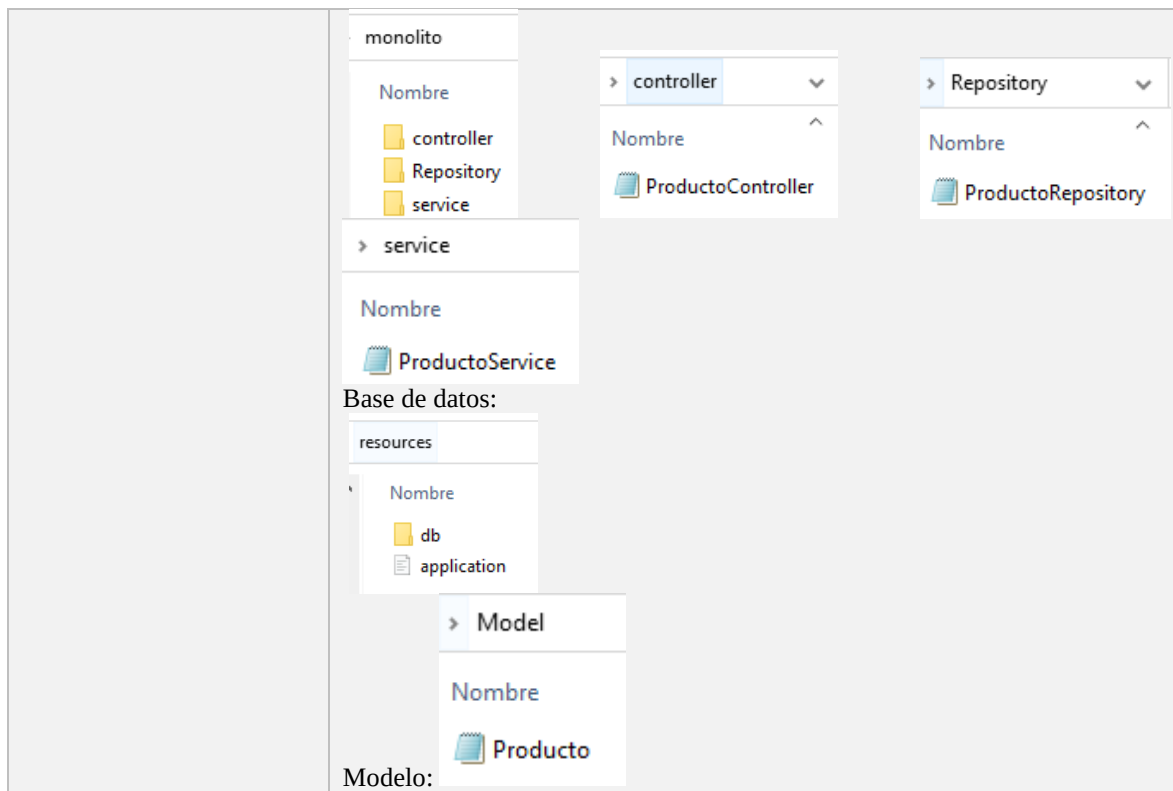
Modelo:

monito > model

Nombre

Categoria

	monolito Nombre controller Repository service service Nombre DetallePedidoService Base de datos: db Nombre db application Modelo: Model Nombre DetallePedido controller Nombre DetallePedidoController Repository Nombre ClientePedidoRepository
Microservicio Pedido	Cap. de Negocio: monolito Nombre controller Repository service controller Nombre PedidoController Repository Nombre PedidoRepository service Nombre PedidoService Base de datos: resources Nombre db application Modelo: Model Nombre Pedido
Microservicio Producto	Cap. de Negocio:

			
	Resultado de la prueba	La herramienta genera de forma correcta las bases de datos por cada microservicio, así mismo, el modelo de datos (tablas) correspondiente a la capacidad de negocio del microservicio es generado de forma correcta.	

A continuación, en las tablas 24, 25 y 26 se muestran los resultados de la ejecución del diseño de prueba “ISMONO2MSDP0201” para 3 casos de uso sobre el caso de prueba del sistema de gestión de ventas “ISMONO2MSCP02”.

Tabla 24 Evaluación del funcionamiento del caso de uso CategoriaControllerTest.

Nombre de la prueba	ISMONO2MSCP01
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de ventas
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	CategoriaControllerTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	Pruebas realizadas sin errores / Categoría guardada con ID asignado

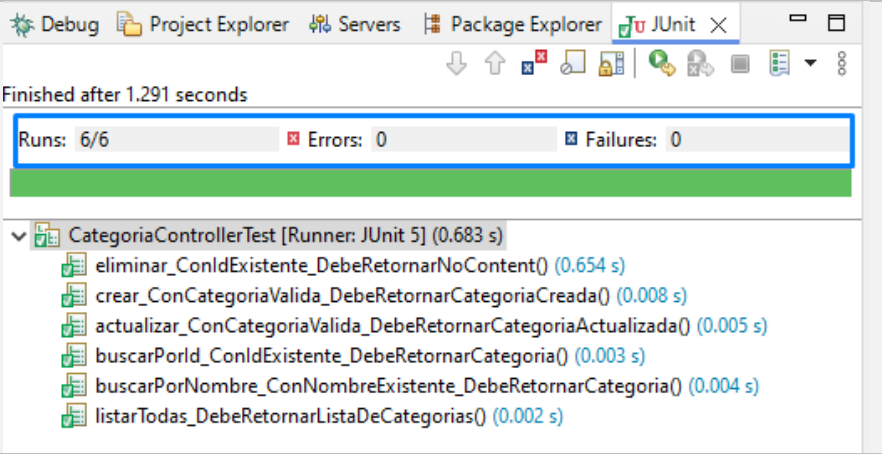
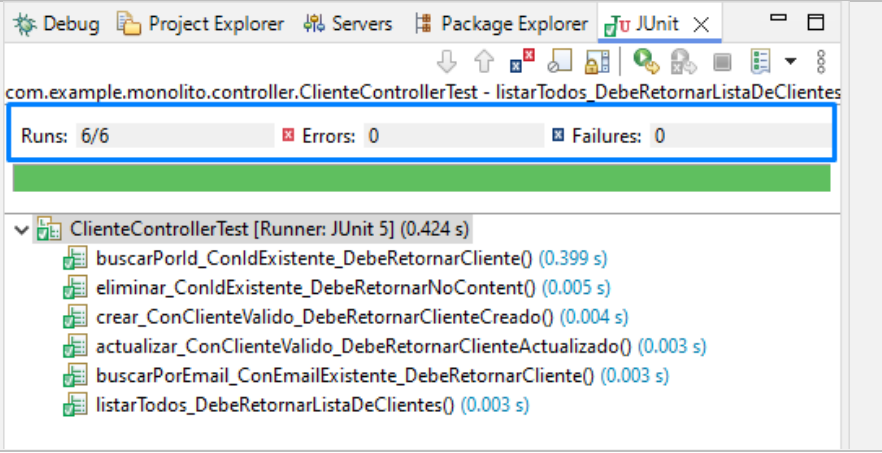
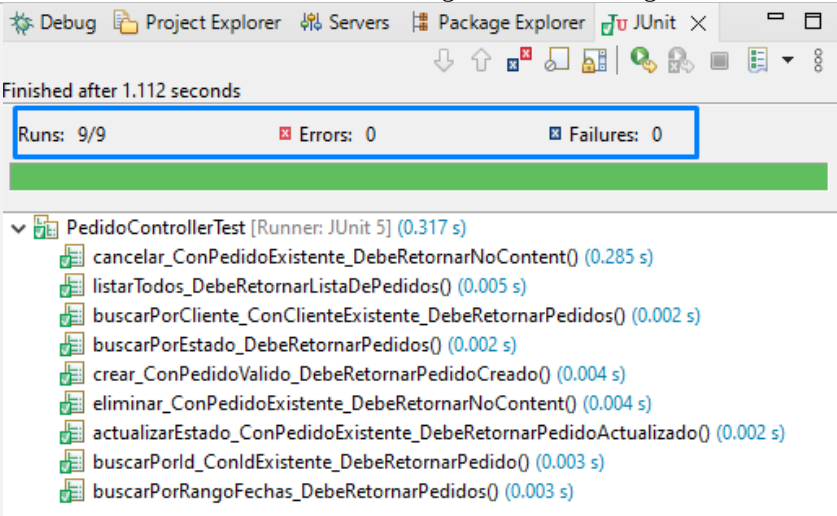
Resultado obtenido (Monolito)	
Estado Monolito	✓ PASA
Resultado obtenido (Microservicio)	Categoria { "id": 1, "nombre": "Relojes", "descripcion": "Digital", "productos": [{ "id": 101,"nombre": "Casio F91W","precio": 499.99 },] },}— HTTP 201
Estado Microservicio	✓ PASA
Comando de prueba en monolito	mvn test -pl sample-monolith -Dtest=CustomerServiceTest

Tabla 25 Evaluación del funcionamiento del caso de uso ClienteControllerTest.

Nombre de la prueba	ISMONO2MSCP01
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de ventas
Objetivo	Evaluar el correcto funcionamiento del sistema de legado que implementa el método de descomposición antes y después de su aplicación
Caso de uso	ClienteControllerTest
Datos de entrada	Código fuente del sistema legado monolítico en la herramienta que implementa el método.
Resultado esperado	Pruebas realizadas sin errores / Cliente registrado con ID asignado
Resultado obtenido (Monolito)	
Estado Monolito	✓ PASA

Resultado obtenido (Microservicio)	Cliente { id=1, name='Ana', Apellido='García' email='ana@test.com', telefono=55534565655,direccion='de las rosas 130', 'pedidos': [] } — HTTP 201
Estado Microservicio	✓ PASA
Comando de prueba en monolito	mvn test -pl sample-monolith -Dtest=ClienteControllerTest

Tabla 26 Evaluación del funcionamiento del caso de uso PedidoControllerTest.

Nombre de la prueba	ISMONO2MSCP01
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de ventas
Objetivo	Evaluar el correcto funcionamiento del sistema de legado que implementa el método de descomposición antes y después de su aplicación
Caso de uso	PedidoControllerTest
Datos de entrada	Código fuente del sistema legado monolítico en la herramienta que implementa el método.
Resultado esperado	Pruebas realizadas sin errores / Pedido registrado con ID asignado
Resultado obtenido (Monolito)	
Estado Monolito	✓ PASA
Resultado obtenido (Microservicio)	Pedido { id=1, estado='PENDIENTE', total=1250.50, fecha='2026-06-15', cliente=Cliente { id=10, nombre='Juan Pérez', email='juan@test.com'}, detalles=[DetallePedido { id=100, cantidad=2, producto='Casio F91W' },] } — HTTP 201
Estado Microservicio	✓ PASA
Comando de prueba en monolito	mvn test -pl sample-monolith -Dtest=PedidoControllerTest

Caso de prueba ISMONO2MSCP02 – Sistema de gestión de facturación

El sistema monolítico objeto de prueba integra funcionalidades relacionadas con la gestión de facturación, a través de los controladores InvoiceController, DiscountController y ReturnController. Estas funcionalidades permiten generar y administrar facturas, aplicar descuentos vigentes sobre el monto facturado, así como procesar devoluciones asociadas a facturas previamente pagadas. A partir de estas operaciones se generan interacciones entre clases y el modelo de datos, cuyas dependencias se representan en el siguiente diagrama de clases en la figura 11.

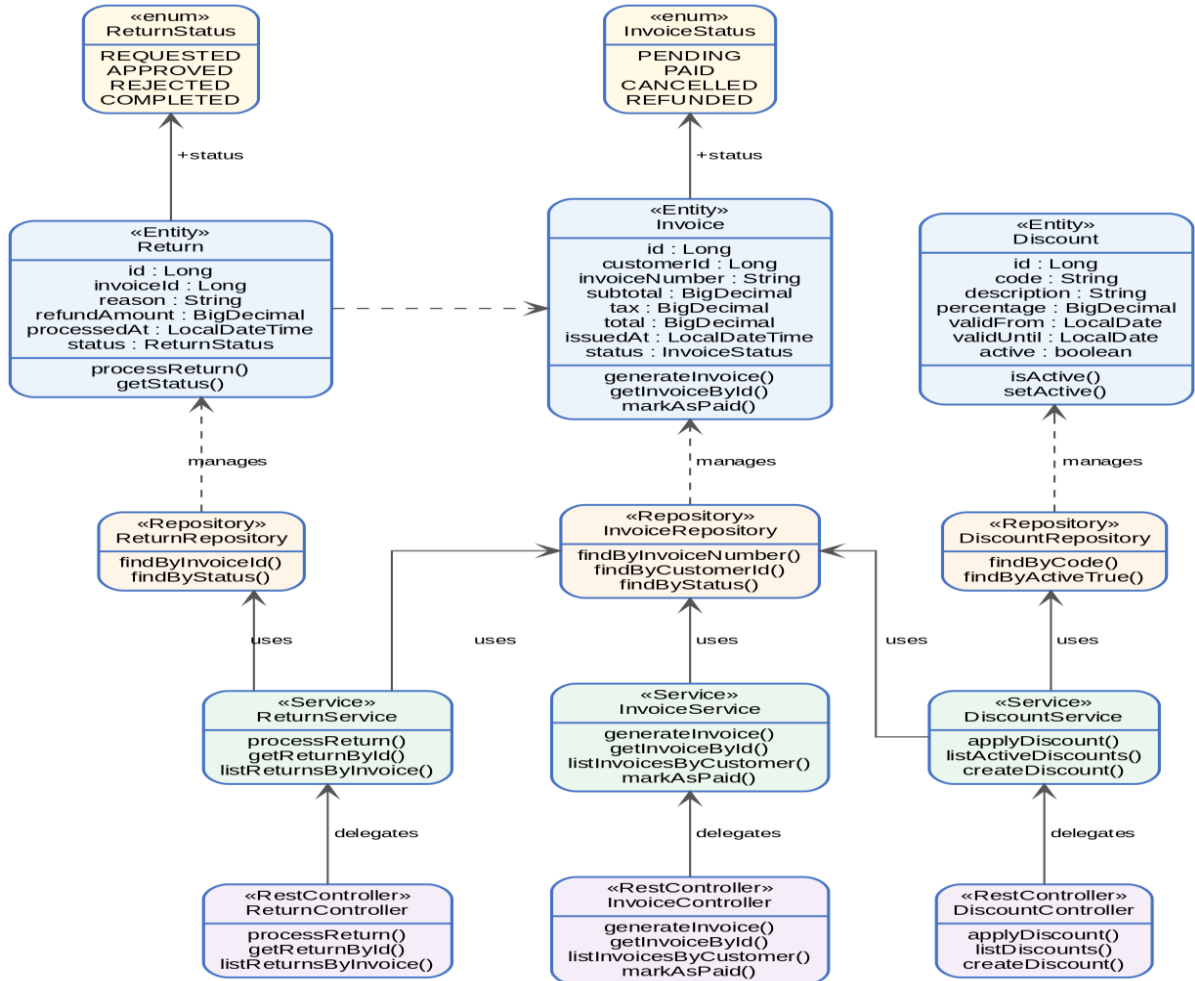


Figura 11 Diagrama de clases del sistema de facturación

En el diagrama de clases se pueden observar las relaciones entre los modelos de datos, y con base en el análisis del código fuente se obtuvieron las siguientes relaciones entre los casos de uso (controladores) y el modelo de datos:

- InvoiceController → modelo Invoice
- DiscountController → modelos Discount e Invoice (a través de DiscountService)
- ReturnController → modelos Return e Invoice (a través de ReturnService)

Estas clases representan capacidades de negocio que interactúan con entidades del modelo de datos y sus tablas. A diferencia del sistema de ventas, en este monolito las entidades no definen relaciones @ManyToOne entre sí mediante JPA; en cambio, la entidad Return almacena el identificador de Invoice como clave foránea lógica (Long). La relación entre servicios y repositorios revela las dependencias cruzadas entre entidades. Revisando el código se obtuvieron las siguientes relaciones:

Tabla 27 Relaciones entre las entidades del sistema de gestión de facturación.

Modelo	Relacionado con	Relación típica (JPA)
Return	Invoice	FK lógica: invoiceId (Long)
DiscountService	Invoice + Discount	acceso multi-repositorio
ReturnService	Invoice + Return	acceso multi-repositorio
Invoice	InvoiceStatus	@Enumerated(EnumType.STRING)

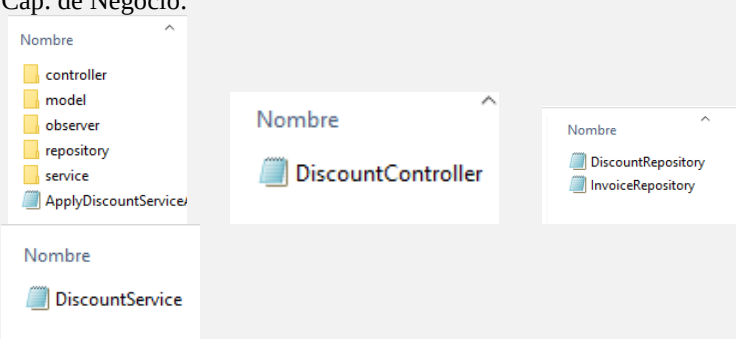
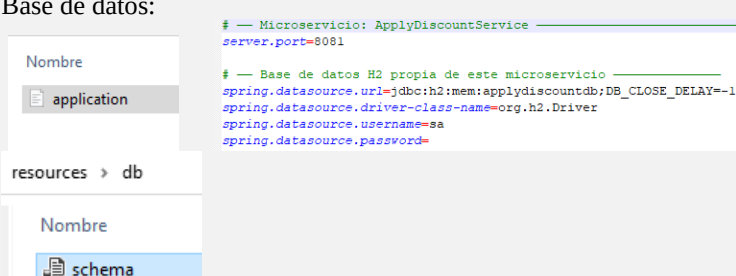
Tabla 28 Relación de casos de uso con el modelo de datos del sistema de gestión de facturación.

Capacidad de negocio	Invoice	Discount	Return
InvoiceController	✓		
DiscountController	✓	✓	
ReturnController	✓		✓

Procedimiento de Ejecución de la prueba

Los resultados de la ejecución del plan de pruebas realizada al sistema de gestión de facturación se muestran en las tablas 29, 30, 31 y 32 descritas a continuación.

Tabla 29 Generación de bases de datos por microservicio para el sistema de gestión de facturación.

Nombre de la prueba	ISMONO2MSCP02
Diseño de prueba	ISMONO2MSDP01
Sistema legado (monolito)	Sistema de gestión de facturación
Entrada	Código fuente del sistema legado monolítico en la herramienta que implementa el método.
Objetivo	Verificar que la base de datos de cada microservicio y su estructura de tablas haya sido generada correctamente.
ApplyDiscountService	Cap. de Negocio:  Base de datos:  <pre># --- Microservicio: ApplyDiscountService --- server.port=8081 # --- Base de datos H2 propia de este microservicio --- spring.datasource.url=jdbc:h2:mem:applydiscountdb;DB_CLOSE_DELAY=-1 spring.datasource.driver-class-name=org.h2.Driver spring.datasource.username=sa spring.datasource.password=</pre>

	Modelo: Nombre Discount Invoice
ReturnControllerService	Cap. de Negocio: Nombre ReturnController Nombre ReturnService Nombre InvoiceRepository ReturnRepository Base de datos: <pre># -- Microservicio: ProcessReturnService -- server.port=8088 # -- Base de datos H2 propia de este microservicio -- spring.datasource.url=jdbc:h2:mem:processreturndb; spring.datasource.driver-class-name=org.h2.Driver spring.datasource.username=sa</pre> Modelo: Nombre Invoice Return
GenerateInvoiceService	Negocio: Nombre controller model observer repository service GenerateInvoiceServiceApplication Nombre InvoiceRepository Nombre InvoiceService Nombre InvoiceController Base de datos: <pre># -- Microservicio: GenerateInvoiceService -- server.port=8084 # -- Base de datos H2 propia de este microservicio -- spring.datasource.url=jdbc:h2:mem:generateinvoicedb; spring.datasource.driver-class-name=org.h2.Driver spring.datasource.username=sa</pre> Modelo: Nombre Invoice
Resultado de la prueba	La herramienta genera de forma correcta las bases de datos por cada microservicio, así mismo, el modelo de datos (tablas) correspondiente a la capacidad de negocio del microservicio es generado de forma correcta.

Tabla 30 Evaluación del funcionamiento del caso de uso InvoiceServiceTest

Nombre de la prueba	ISMONO2MSCP02
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de facturación
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	InvoiceServiceTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Factura generada con número INV-XXXX automático y total = subtotal + tax
Resultado obtenido (Monolito)	[INFO] Running com.example.billing.service.InvoiceServiceTest [INFO] TC-M2-01: Sin número -> genera INV-XXXX automáticamente PASSED [INFO] TC-M2-02: Con número manual -> conserva número existente PASSED [INFO] TC-M2-03: Total = subtotal + tax calculado correctamente PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.312 s [INFO] BUILD SUCCESS
Estado Monolito	OK PASA
Resultado obtenido (Microservicio)	POST /api/invoices -> HTTP 201 { "id": 1, "invoiceNumber": "INV-A1B2C3D4", "customerId": 1, "subtotal": 5000.00, "tax": 800.00, "total": 5800.00, "issuedAt": "2026-06-14T10:00:00", "status": "PENDING" }
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-2 -Dtest=InvoiceServiceTest

Tabla 31 Evaluación del funcionamiento del caso de uso DiscountServiceTest.

Nombre de la prueba	ISMONO2MSCP02
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de facturación
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	DiscountServiceTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Descuento 10% aplicado (subtotal=90, tax=14.40, total=104.40); inválidos y expirados rechazados
Resultado obtenido (Monolito)	[INFO] Running com.example.billing.service.DiscountServiceTest [INFO] TC-M2-07: Descuento 10% vigente -> total reducido correctamente PASSED [INFO] TC-M2-08: Código inexistente -> lanza RuntimeException PASSED [INFO] TC-M2-09: Descuento expirado -> lanza IllegalStateException PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.198 s [INFO] BUILD SUCCESS
Estado Monolito	OK PASA

Resultado obtenido (Microservicio)	POST /api/discounts/apply?invoiceId=1&code=PROMO10 -> HTTP 200 <pre>{ "id": 1, "invoiceNumber": "INV-001", "subtotal": 90.00, "tax": 14.40, "total": 104.40, "status": "PENDING" }</pre> (10% sobre subtotal=100.00 + tax=16.00 originales)
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-2 -Dtest=DiscountServiceTest

Tabla 32 Evaluación del funcionamiento del caso de uso ReturnServiceTest.

Nombre de la prueba	ISMONO2MSCP02
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de facturación
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	ReturnServiceTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Devolución APPROVED sobre factura PAID; factura -> REFUNDED; PENDING y monto excedido rechazados
Resultado obtenido (Monolito)	[INFO] Running com.example.billing.service.ReturnServiceTest [INFO] TC-M2-04: Factura PAGADA -> devolución APPROVED + factura REFUNDED PASSED [INFO] TC-M2-05: Factura PENDING -> lanza IllegalStateException PASSED [INFO] TC-M2-06: Monto > total factura -> lanza IllegalArgumentException PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.154 s [INFO] BUILD SUCCESS
Estado Monolito	OK PASA
Resultado obtenido (Microservicio)	POST /api/returns -> HTTP 201 <pre>{ "id": 1, "invoiceId": 1, "reason": "Producto dañado", "refundAmount": 116.00, "status": "APPROVED", "processed": "2026-06-14T10:30:00" }</pre> Invoice.status actualizado a REFUNDED
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-2 -Dtest=ReturnServiceTest

Caso de prueba ISMONO2MSCP03 – Sistema de gestión de fidelización

El sistema monolítico objeto de prueba integra funcionalidades relacionadas con la gestión del programa de fidelización de clientes, a través de los controladores LoyaltyController y RewardController. Estas funcionalidades permiten inscribir clientes en el programa, acumular puntos por compras y canjear recompensas deduciendo el saldo correspondiente. A partir de estas operaciones se generan interacciones entre clases y el modelo de datos, cuyas dependencias se representan en el siguiente diagrama de clases.

En el diagrama de clases de la figura 12, se pueden observar las relaciones entre los modelos de datos y con base en el análisis del código fuente se obtuvieron las siguientes relaciones entre los casos de uso (controladores) y el modelo de datos:

- LoyaltyController → modelos LoyaltyAccount y PointTransaction (a través de LoyaltyService)
- RewardController → modelos Reward, LoyaltyAccount y PointTransaction (a través de RewardService)

Estas clases representan capacidades de negocio que interactúan con entidades del modelo de datos. Al igual que en el sistema de facturación, las entidades no utilizan @ManyToOne entre sí; la entidad PointTransaction almacena el identificador de LoyaltyAccount como clave foránea lógica (Long). El servicio RewardService accede a tres repositorios distintos para coordinar el canje de recompensas con la deducción de puntos. Esta interacción se puede observar en la figura 12.

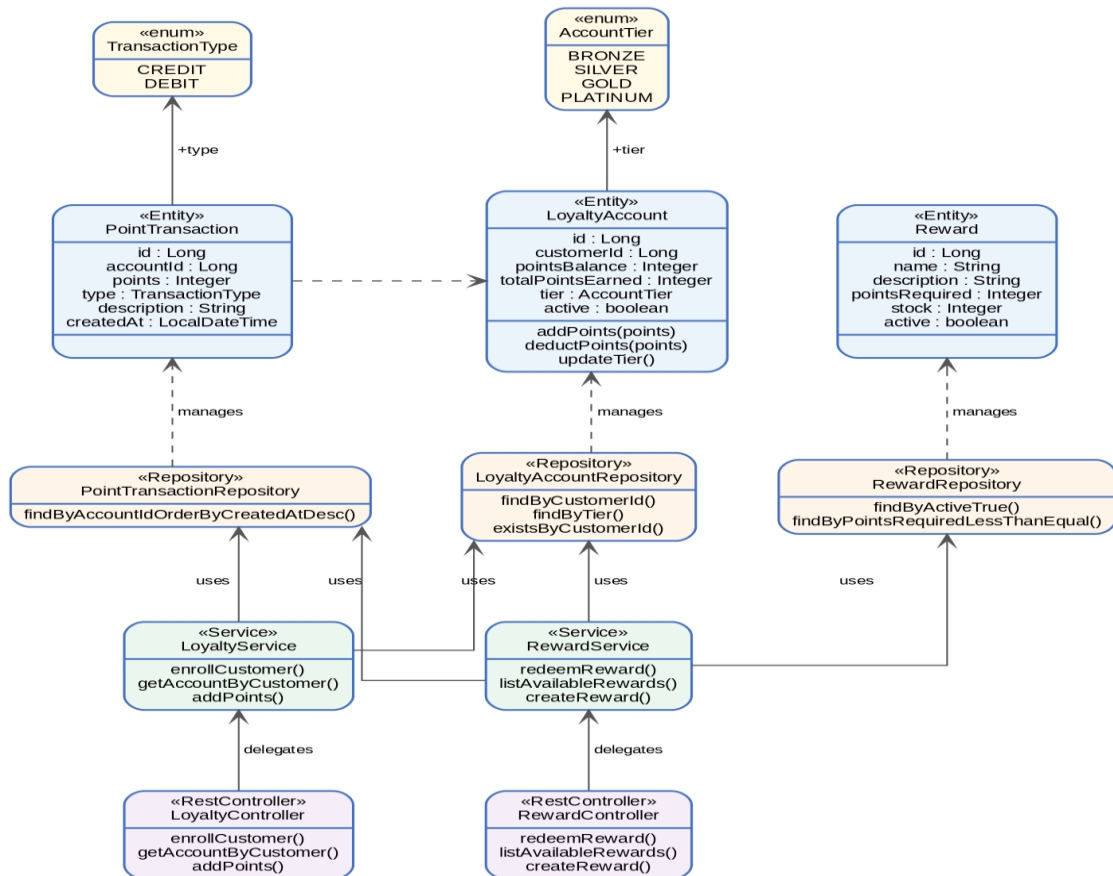


Figura 12 Diagrama de clases del sistema de gestión de fidelización

La relación entre servicios y repositorios revela las dependencias cruzadas entre entidades. En la tabla 33 y 34 se pueden observar las siguientes relaciones entre las entidades del modelo del sistema de fidelización:

Tabla 33 Relaciones entre las entidades del modelo

Modelo	Relacionado con	Relación típica (JPA)
PointTransaction	LoyaltyAccount	FK lógica: accountId (Long)
LoyaltyService	LoyaltyAccount + PointTransaction	acceso multi-repositorio
RewardService	LoyaltyAccount + PointTransaction + Reward	acceso multi-repositorio
LoyaltyAccount	AccountTier	@Enumerated(EnumType.STRING)
PointTransaction	TransactionType	@Enumerated(EnumType.STRING)

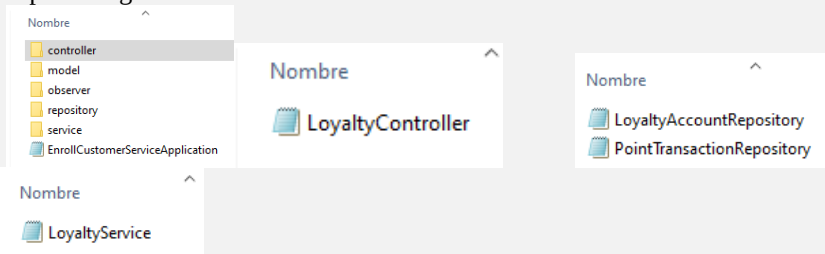
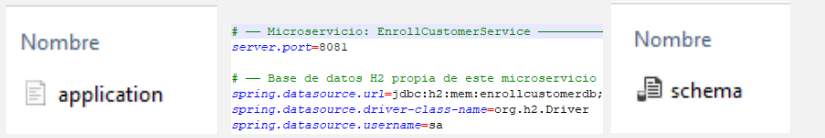
Tabla 34 Relaciones entre las entidades del sistema de gestión de fidelización.

Capacidad de negocio	LoyaltyAccount	PointTransaction	Reward
LoyaltyController	✓	✓	
RewardController	✓	✓	✓
AddPointService	✓	✓	

Procedimiento de Ejecución de la prueba

Los resultados de la ejecución del plan de pruebas realizada en el sistema de gestión de fidelización se muestran en las tablas 35, 36, 37 y 38.

Tabla 35 Generación de DB por microservicio para el sistema de gestión de fidelización.

Nombre de la prueba	ISMONO2MSCP03
Diseño de prueba	ISMONO2MSDP01
Sistema legado (monolito)	Sistema de gestión de fidelización
Entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Objetivo	Verificar que la base de datos de cada microservicio y su estructura de tablas han sido generada correctamente.
EnrollCustomerService	Cap. de Negocio:  Base de datos:  Modelo:

	Nombre LoyaltyAccount PointTransaction
CreateRewardService	Cap. de Negocio: Nombre controller model observer repository service CreateRewardServiceApplication Nombre LoyaltyAccountRepository PointTransactionRepository RewardRepository Nombre RewardController Nombre RewardService Base de datos: Microservicio: CreateRewardService server.port=8086 Base de datos H2 propia de este microservicio spring.datasource.url=jdbc:h2:mem:createrewarddb;1 spring.datasource.driver-class-name=org.h2.Driver spring.datasource.username=sa Nombre schema Modelo: Nombre LoyaltyAccount PointTransaction Reward
AddPointService	Negocio: Nombre controller model observer repository service AddPointsServiceApplication Nombre LoyaltyController Nombre LoyaltyAccountRepository PointTransactionRepository Nombre LoyaltyService Base de datos: Microservicio: AddPointsService server.port=8083 Base de datos H2 propia de este microservicio spring.datasource.url=jdbc:h2:mem:addpointssdb; spring.datasource.driver-class-name=org.h2.Driver spring.datasource.username=sa Nombre schema Modelo: Nombre LoyaltyAccount PointTransaction
Resultado de la prueba	La herramienta genera de forma correcta las bases de datos por cada microservicio, así mismo, el modelo de datos (tablas) correspondiente a la capacidad de negocio del microservicio es generado de forma correcta.

Sistema de gestión de fidelización

Tabla 36 Evaluación del funcionamiento del caso de uso ReturnServiceTest.

Nombre de la prueba	ISMONO2MSCP03
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de fidelización
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	LoyaltyServiceEnrollTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Cuenta creada con 0 puntos y tier BRONZE activa; inscripción duplicada rechazada
Resultado obtenido (Monolito)	[INFO] Running com.example.loyalty.service.LoyaltyServiceEnrollTest [INFO] TC-M3-01: Cliente nuevo -> cuenta BRONZE con 0 puntos creada PASSED [INFO] TC-M3-02: Cliente ya inscrito -> lanza IllegalStateException PASSED [INFO] TC-M3-03: Cuenta nueva queda activa por defecto PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.221 s [INFO] BUILD SUCCESS
Estado Monolito	OK PASA
Resultado obtenido (Microservicio)	POST /api/royalty/enroll/10 -> HTTP 201 { "id": 1, "customerId": 10, "pointsBalance": 0, "totalPointsEarned": 0, "tier": "BRONZE", "active": true }
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-3 -Dtest=LoyaltyServiceEnrollTest

Tabla 37 Evaluación del funcionamiento del caso de uso LoyaltyServiceEnrollTest.

Nombre de la prueba	ISMONO2MSCP03
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de fidelización
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	LoyaltyServiceAddPointsTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Puntos sumados, transacción CREDIT registrada; 1000+ puntos -> tier SILVER; puntos <= 0 rechazados
Resultado obtenido (Monolito)	[INFO] Running com.example.loyalty.service.LoyaltyServiceAddPointsTest [INFO] TC-M3-07: Puntos validos -> saldo actualizado y transacción CREDIT PASSED [INFO] TC-M3-08: 1000 puntos acumulados -> tier sube a SILVER PASSED [INFO] TC-M3-09: Puntos = 0 -> lanza IllegalArgumentException PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.177 s

	[INFO] BUILD SUCCESS
Estado Monolito	OK PASA
Resultado obtenido (Microservicio)	POST /api/royalty/customer/5/points?points=300&description=Compra -> HTTP 200 <pre>{ "id": 1, "customerId": 5, "pointsBalance": 300, "totalPointsEarned": 300, "tier": "BRONZE" }</pre> PointTransaction { "type": "CREDIT", "points": 300 }
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-3 -Dtest=LoyaltyServiceAddPointsTest

Tabla 38 Evaluación del funcionamiento del caso de uso LoyaltyServiceAddPointsTest.

Nombre de la prueba	ISMONO2MSCP03
Diseño de prueba	ISMONO2MSDP02
Sistema legado (monolito)	Sistema de gestión de fidelización
Objetivo	Evaluar el correcto funcionamiento del sistema legado y los microservicios generados antes y después de la aplicación del método de descomposición en la herramienta informática.
Caso de uso	RewardServiceRedeemTest
Datos de entrada	Código fuente monolítico del sistema legado en la herramienta que implementa el método.
Resultado esperado	3/3 pruebas sin errores / Recompensa canjeada; puntos deducidos (5000-200=4800); stock 10->9; puntos insuficientes y sin stock rechazados
Resultado obtenido (Monolito)	[INFO] Running com.example.loyalty.service.RewardServiceRedeemTest [INFO] TC-M3-04: Puntos suficientes y stock -> canje exitoso, puntos deducidos PASSED [INFO] TC-M3-05: Puntos insuficientes -> lanza IllegalStateException PASSED [INFO] TC-M3-06: Sin stock -> lanza IllegalStateException PASSED [INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.143 s [INFO] BUILD SUCCESS
Estado Monolito	OK PASA
Resultado obtenido (Microservicio)	POST /api/rewards/redeem?customerId=1&rewardId=1 -> HTTP 200 <pre>{ "id": 1, "name": "Café gratis", "pointsRequired": 200, "stock": 9, "active": true }</pre> LoyaltyAccount.pointsBalance: 4800 (5000 - 200 canjeados) PointTransaction { "type": "DEBIT", "points": 200 }
Estado Microservicio	OK PASA
Comando de prueba en monolito	mvn test -pl sample-monolith-3 -Dtest=RewardServiceRedeemTest

4.3. Evaluación experimental del método

El experimento tiene por objetivo validar que la aplicación del método de descomposición propuesto reduce el acoplamiento indirecto originado por compartir una base de datos monolítica. Se compara el grado de acoplamiento antes (sistema monolítico) y después (microservicios generados por el método) en una muestra representativa de sistemas monolíticos legados, obtenidos desde repositorios públicos. La unidad primaria de análisis es el repositorio²; en cada repositorio se analizan las secuencias de casos de uso y se calcula un índice global de acoplamiento basado en accesos a tablas compartidas.

En este contexto, el presente estudio experimental tiene como objetivo evaluar la efectividad del método propuesto para disminuir el acoplamiento al migrar el monolito hacia una arquitectura de microservicios y descomponer la base de datos monolítica, identificando posibles candidatos a microservicios y sus bases de datos por servicio mediante el análisis de la secuencia interactiva entre clases, métodos y entidades que participan en la atención de una meta de valor (caso de uso). El experimento busca validar si la aplicación del método permite disminuir el acoplamiento al identificar unidades funcionales coherentes, con bajo acoplamiento y con límites de servicio alineados con los patrones de migración arquitectónica (Chen et al., 2017).

Es importante señalar que el método propuesto no busca reconstruir el sistema monolítico original. En su lugar, produce como resultado un conjunto de microservicios independientes, cada uno con su propia base de datos, definidos a partir de la secuencia interactiva de casos de uso que responden a una meta de valor específica.

4.3.1 Metodología empleada

En el campo del diseño experimental, (Hinkelmann & Kempthorne, 2007) establecen una metodología sistemática para la construcción de modelos experimentales que permite garantizar la validez estadística y la rigurosidad científica de los estudios.

Este estudio adopta esta metodología como referencia principal para el diseño del modelo experimental, con el objetivo de garantizar que las decisiones sobre el tamaño de la muestra, la selección de unidades y el análisis de los datos se realicen bajo un esquema reconocido y validado en la literatura estadística. De esta manera, se asegura que el experimento no solo responda a la pregunta de investigación planteada, sino que también cumpla con los estándares de calidad y confiabilidad exigidos en la investigación científica.

El modelo utilizado en este estudio se centra en la definición de:

- Preguntas de investigación e hipótesis.

² En este documento, la palabra “repositorio” se refiere a los sistemas monolíticos legados, que actualmente se encuentran en operación, y que fueron obtenidos a través de la búsqueda en repositorios web “GITHUB” y considerados en las pruebas.

-
- La identificación de las variables relevantes.
 - Diseño de las pruebas.
 - Desarrollo del modelo estadístico.
 - Ejecución del experimento.
 - Recolección de los datos.
 - Análisis estadístico.
 - Interpretación de los resultados.

4.3.2 Pregunta de investigación e hipótesis

Pregunta de investigación: ¿Un método de descomposición de una base de datos global para cada microservicio generado después de su aplicación reduce de forma significativa el acoplamiento indirecto causado por compartir la base de datos en sistemas monolíticos?

Formulación estadística:

- **Hipótesis nula (H_0):** La implementación del método de migración de sistemas monolíticos hacia microservicios, que incluye la descomposición de la base de datos global, no produce una reducción del Índice de Acoplamiento entre Tablas Compartidas (IATC) entre el estado previo y el estado posterior a la migración.
- **Hipótesis alternativa (H_1):** La implementación del método de migración de sistemas monolíticos hacia microservicios, que incluye la descomposición de la base de datos global, produce una reducción del Índice de Acoplamiento entre Tablas Compartidas (IATC) entre el estado previo y el estado posterior a la migración.

Para contrastar las hipótesis planteadas, se realiza la evaluación por pares (cada proyecto actúa de forma independiente con sus correspondientes casos de uso: antes vs después de la aplicación del método).

4.3.3 Variables dependientes e independientes

Variables independientes:

- Arquitectura Estado arquitectónico del sistema: configuración inicial del monolito con base de datos compartida y configuración resultante de la migración hacia microservicios con bases de datos aisladas por servicio.
- Reglas de asignación de entidades y tablas a microservicios candidatos, derivadas de las secuencias de interacción de los casos de uso.
- Sincronización de datos replicados entre servicios mediante el patrón Observador.
- Complejidad del caso de uso: CRUD simple, transacción multi-entidad entre tablas compartidas.

Variables dependientes:

- Índice de acoplamiento de tablas compartidas (IATC), ecuación 1:

$$IATC = \frac{accTab - NUT}{|NCU - (NUT * NCU)|} \quad (1)$$

El IATC se calcula a partir de:

- accTab: Total de accesos únicos a las tablas de la base de datos.
- NUT: Número total de tablas.
- NCU: Número total de casos de uso.

4.3.4 Diseño del plan de pruebas

En este estudio se aplicará un nivel de prueba de calidad, utilizando como tipo de prueba la métrica IATC. En concordancia con la metodología experimental planteada por (Hinkelmann & Kempthorne, 2007), el presente estudio se estructura en una serie de fases que permiten garantizar la validez estadística y la rigurosidad científica del modelo experimental. En primer lugar, se procede a la selección de sistemas monolíticos con base de datos integrada, los cuales son obtenidos de repositorios públicos GitHub en la plataforma Git y constituyen los casos de estudio sobre los que se aplicará el método de descomposición propuesto. Posteriormente, se lleva a cabo la aplicación del método de descomposición, que consiste en migrar cada sistema monolítico hacia microservicios independientes, uno por cada caso de uso identificado, con su correspondiente descomposición de su base de datos autónoma.

Con la finalidad de contrastar el factor de acoplamiento, el cálculo de la métrica de calidad se efectúa antes de la migración del sistema monolítico hacia microservicios, los resultados obtenidos se registran en una matriz de datos estructurada. Una vez realizada la migración, nuevamente se efectúa el cálculo de la métrica de calidad, orientada a medir el factor de acoplamiento entre los microservicios generados por la descomposición de la base de datos. Así mismo, los resultados obtenidos se registran en la matriz de datos, lo que permite comparar de manera sistemática el estado inicial (monolito) y el estado resultante (microservicios).

En la siguiente fase, el tamaño de cada repositorio se organiza por estratos, determinados por el número de casos de uso identificados, como se describe en la sección 5.3 de este documento. Posteriormente, se aplican técnicas estadísticas descriptivas e inferenciales con el fin de verificar la reducción del acoplamiento y evaluar la frecuencia relativa de los factores de calidad en los distintos estratos. Finalmente, se realiza la interpretación de resultados, en la cual se contrastan los hallazgos con las hipótesis planteadas, identificando patrones relevantes y extrayendo conclusiones sobre la efectividad del método de descomposición en la calidad de los sistemas analizados.

4.3.5 Enfoque y Criterios de Aceptación

El criterio de aprobación o desaprobación de las pruebas de calidad será mediante la comparación de los resultados obtenidos al aplicar el cálculo de la métrica, inicialmente, de forma manual antes y después de la aplicación del método de descomposición.

Criterios de aceptación:

Se aplica la métrica formal de acoplamiento.

- Comparación cuantitativa entre el nivel de acoplamiento en el sistema monolítico original correspondiente al repositorio y en los microservicios obtenidos de manera apropiada.
- Debe comprobarse una disminución sobre el acoplamiento entre los microservicios generados en comparación al sistema monolito de origen.

4.3.6 Diseño de la prueba

El objetivo del plan de pruebas es demostrar que el acoplamiento indirecto entre los microservicios, causado por compartir la base de datos en un sistema legado correspondiente al repositorio en operación ha sido reducido. La evaluación se realiza mediante la métrica IATC. Esto permitirá confirmar que el proceso de descomposición ha logrado disminuir la dependencia entre los servicios del monolito original con su base de datos global al ser migrado hacia microservicios, con su única base de datos y sus tablas relacionadas.

Para este experimento, se realizó la validación del método solo para repositorios en lenguaje Java y con SpringBoot, JPA y ORM, debido a que la herramienta de pruebas está realizada para cumplir con esa especificación, independientemente de que el método aplique para cualquier sistema monolítico.

Los pasos del plan de pruebas son los siguientes:

1.-Selección de los sistemas legados en operación.

Se realizó una búsqueda de repositorios monolíticos que cumplan con las limitaciones y alcances de esta investigación, a través de una búsqueda en repositorios públicos de control de versiones utilizando la plataforma GitHub, accesible en el siguiente enlace: <https://github.com/>.

2.- Diseño de los casos de prueba.

Para este plan de pruebas se realizó el diseño de prueba “Reducción del acoplamiento Indirecto” el cual permite validar las hipótesis planteadas en la sección 5.1 de este documento, la descripción del diseño de pruebas se presenta en la tabla 39.

Tabla 39 Diseño de prueba “Reducción del acoplamiento indirecto”.

Diseño de Prueba - Reducción del Acoplamiento indirecto	
Identificador	ISMONO2MSDP03
Característica a ser probada	Evaluación de la disminución del acoplamiento entre microservicios obtenidos después de la aplicación del método de migración.
Descripción	Demostrar que se reducen las dependencias entre los microservicios derivados del sistema legado, tras la migración y descomposición de su correspondiente modelo de datos.
Criterios de Aprobación	El acoplamiento debe ser mayor entre los microservicios derivados de un monolito original, que aún comparten datos globales, contra los microservicios obtenidos después de la descomposición del modelo de datos. En tal caso, se puede afirmar que cada microservicio es independiente y autónomo.
Métrica de Evaluación	La métrica calcula el número de dependencias indirectas por el modelo de datos entre los microservicios, antes y después de la descomposición.
Refinamiento	Evaluar que las relaciones entre microservicios sean mínimas, especialmente aquellas que dependen del modelo de datos. Comprobar que cada microservicio sea autónomo y no dependa de otros para acceder a su base de datos y sus tablas.
Resultado Esperado	El acoplamiento indirecto entre los servicios del monolito original, debe ser reducido al aplicar el método de descomposición de la base de datos.

3. Cálculo de métricas de calidad estructural

Los repositorios seleccionados constituyen los instrumentos de prueba. A cada uno se le aplicará la métrica IATC. Esta métrica permitirá cuantificar objetivamente si el índice de acoplamiento fue reducido.

4. Registro y consolidación de resultados

Los resultados obtenidos, mediante la aplicación de la fórmula para calcular el vector de diferencias y validar si los datos están normalizados, se registrarán en una matriz de datos estructurada.

5. Análisis estadístico

Se aplicarán técnicas estadísticas descriptivas para validar las hipótesis.

6. Interpretación de resultados

Se realiza la interpretación de los resultados y se presentan los gráficos y Boxplot derivados de las pruebas aplicadas.

4.3.7 Caso de prueba

Para la validación del método propuesto fue necesario realizar una búsqueda de repositorios en la plataforma web GitHub que cumplieran con criterios específicos relacionados con tecnologías de desarrollo en Java. Para ello, se utilizó la siguiente cadena de búsqueda en la API de GitHub:

[https://api.github.com/search/repositories?q=language:Java+\"Spring+Boot\"+\"Spring+Data+JPA\"+hibernate+database&per_page=100&page=1&sort=stars&order=desc](https://api.github.com/search/repositories?q=language:Java+\)

En una primera etapa, la consulta se ejecutó mediante la herramienta Postman arrojando un total de 80 repositorios, lo que permitió verificar la correcta construcción de la petición y validar la respuesta obtenida desde la API. Posteriormente, con el objetivo de automatizar el proceso y hacerlo dinámico, se implementó un programa en Python que permitió extraer de manera automática los repositorios encontrados, incluyendo información relevante como el nombre del repositorio, su descripción y la URL de acceso al mismo.

Una vez obtenidos los resultados, se exportaron a un archivo en formato CSV para facilitar su análisis. En esta fase se accedió a la url de los repositorios y fueron revisados con el fin de descartar aquellos que no cumplían con las condiciones esperadas, por ejemplo, la falta de implementación de las tecnologías requeridas, no contaban con pruebas unitarias, el número de clases y casos de uso eran casi nulos, así como la ausencia de documentación suficiente.

Tras este filtrado, se conformó un universo final de 20 repositorios, los cuales fueron considerados adecuados para la evaluación del método, estos son descritos a continuación:

1. **Springboot-thymeleaf-crud-pagination-sorting-webapp** Aplicación web CRUD en Spring Boot con funciones de paginación y ordenamiento, utilizando Spring Boot, Thymeleaf, Spring Data JPA, Hibernate y base de datos MySQL.
2. **spring-boot-hibernate-oracle** Ejemplo de aplicación con Spring Boot, Hibernate y Oracle que incluye conexión a base de datos y uso de Spring Data JPA para exportar APIs REST.
3. **Springboot-blog-rest-api** Proyecto: Aplicación de Blog usando Spring Boot con APIs REST en tiempo real, seguridad con Spring Security, autenticación basada en roles, JWT, Spring Data JPA (Hibernate), paginación y base de datos MySQL. Incluye documentación de API con Swagger y uso de DTO para transferencia de datos. Este proyecto es una experiencia inmersiva en la creación de una aplicación de blog dinámica con capacidades en tiempo real utilizando el framework Spring. Abarca una amplia gama de conceptos y herramientas, incluyendo Spring Boot, Spring Security, JWT, Spring Data JPA (Hibernate) y base de datos MySQL.
4. **Springboot-thymeleaf-employee-crud** Aplicación web CRUD en Spring Boot para gestión de empleados, con funciones de paginación y ordenamiento, utilizando Spring Boot, Thymeleaf, Spring Data JPA, Hibernate y base de datos MySQL.
5. **SpringBoot-BookManegement** El proyecto de gestión de libros es una API RESTful para administrar libros utilizando Spring Boot, Spring Data JPA y Hibernate. Soporta operaciones CRUD, permitiendo a los usuarios agregar, actualizar, eliminar y recuperar libros. Spring Data JPA simplifica las interacciones con la base de datos, mientras que Hibernate mapea objetos Java a tablas. El proyecto está construido usando MySQL como base de datos.
6. **Spring-Boot-Jpa-Hibernate-One-To-One-Mapping-ExampleApplication** Este repositorio muestra cómo implementar paso a paso un mapeo de entidades uno a uno

usando JPA/Hibernate con Spring Boot, Spring Data JPA y una base de datos PostgreSQL. La anotación @OneToMany de JPA se utiliza para mapear la entidad origen con la entidad destino. En este ejemplo se crean las entidades Instructor e InstructorDetail y se realiza un mapeo uno a uno entre ellas.

7. **SpringBoot-CRUD** Este módulo es una demostración de operaciones CRUD en Spring Boot con manejo de excepciones, utilizando la base de datos MySQL junto con Hibernate ORM y Spring Data JPA.
8. **Horeca** Proyecto en Java con Maven y Spring Boot que incluye APIs, servicios web, servicios REST, Hibernate, base de datos MySQL, conexión JDBC, API de Cloudinary para almacenamiento de imágenes, paginación en el servidor usando Spring Data JPA, búsqueda en el servidor, manejadores de excepciones personalizadas, seguridad JWT, registro de usuarios, confirmación y validación por correo electrónico, inicio de sesión, además de tecnologías como JavaScript, HTML, CSS y Thymeleaf.
9. **LoginReg-SpringBoot-JSP-Java8** Aplicación web de inicio de sesión y registro, creada como demo para fines de estudio diario. Incluye estudio de Java 8, estructuras de datos y algoritmos, llamadas a API, frameworks de colecciones y servicios API como Postman y SoapService (datos enviados en formato XML). Tecnologías utilizadas: Java, Spring Boot, Ajax, jQuery, Bootstrap, Postman, SoapService, Hibernate, Spring Data JPA, base de datos MySQL y PostgreSQL.
10. **library-management-system** Proyecto para construir APIs REST de un sistema de gestión de bibliotecas utilizando Spring Boot, Spring Data JPA (Hibernate) y base de datos MySQL.
11. **Spring-Boot-Registration-and-Login** Proyecto que utiliza Spring Web MVC para la capa web, Spring Data JPA con Hibernate para la capa de acceso a datos, Spring Security para autenticación, inicio y cierre de sesión, Thymeleaf como motor de plantillas, HTML5 y Bootstrap 4 para la interfaz de usuario responsiva, además de JUnit 5 y AssertJ para pruebas unitarias. Base de datos MySQL.
12. **Shopme-ECommerce-Application** “Shopme” es una tienda electrónica de comercio electrónico. Tecnologías utilizadas: Thymeleaf, Bootstrap, jQuery, Java, Spring Boot, Spring Data JPA, Hibernate, Spring Security, Spring OAuth, Spring Mail, JUnit, Mockito, API de PayPal Checkout, base de datos MySQL, despliegue en Heroku y almacenamiento en Amazon S3.
13. **tidb-java-springboot-jpa-quickstart** Una aplicación Java sencilla que se conecta a un clúster de base de datos TiDB y realiza operaciones CRUD básicas con Spring Boot, Spring Data JPA y Hibernate.
14. **SpringBoot-ProductService** Proyecto en Java 8 con Spring Boot, Lombok, Spring Data JPA, Hibernate, API REST y base de datos PostgreSQL.
15. **spring-boot-rest-api-recipes** Una API RESTful simple construida con Spring Boot, Gradle, Hibernate, base de datos H2 y Spring Data JPA.
16. **spring-boot-course-management-system-jpa-hibernate** Proyecto de aplicación Spring Boot diseñado para demostrar la interacción con bases de datos usando Spring Data JPA.
17. **FlashCardSystem** Sistema de tarjetas didácticas desarrollado con Spring Data JPA, Spring Boot y base de datos H2.

-
18. **springboot-thymeleaf-crud-webMVCapp** Aplicación web CRUD en Spring Boot que realiza operaciones de creación, lectura, actualización y eliminación utilizando Thymeleaf, Spring Data JPA, Hibernate y base de datos MySQL.
 19. **LibraryManagementSystem** Proyecto de sistema de gestión de bibliotecas utilizando Spring Boot, Spring Data JPA (Hibernate) y base de datos MySQL.
 20. **school-management** Proyecto de gestión escolar desarrollado con Spring Boot, Spring Data JPA (Hibernate), Spring Security, Thymeleaf y base de datos PostgreSQL.

4.3.8 Aplicación de la métrica

Para validar el método, se desarrolló la métrica “*índice de acoplamiento por tablas compartidas (IATC)*” y se calcula a partir de la diferencia del total de accesos a las tablas de la base de datos realizados por el conjunto de casos de uso o Microservicios, y el número total de tablas existentes. Este resultado se divide entre el valor absoluto de la diferencia entre el número total de módulos y el producto del número de tablas por el número de módulos del sistema. La fórmula correspondiente se expresa en la ecuación 2 de la siguiente manera:

$$IATC = \frac{accTab - NUT}{|NCU - (NUT * NCU)|} \quad (2)$$

Interpretación de los Valores del IATC

- Se establece el límite donde 0 es el valor menos acoplado y > 0 es el índice de acoplamiento.
- Un valor mayor a 0 de IATC indica un mayor nivel de acoplamiento indirecto en el sistema debido a tablas de base de datos compartidas.
- Un valor de $IATC = 0$ implica que no existen tablas compartidas en el sistema (cada tabla es accedida, como máximo, por un único componente). Este es el estado ideal en una arquitectura de microservicios pura, donde cada servicio tiene total propiedad sobre sus tablas.

Sustento Teórico (ecuación 3)

$$IATC = \frac{accTab - NUT}{|NCU - (NUT * NCU)|} \quad (3)$$

Donde:

- IATC: Índice de acceso a tablas compartidas.
- accTab: Total de accesos únicos a las tablas de la base de datos.
- NUT: Número total de tablas.
- NCU: Número total de casos de uso.

Esta fórmula permite capturar no sólo la cantidad de accesos, sino también la redundancia en el uso de las tablas, lo que incrementa el acoplamiento. El resultado de la métrica tiende a 0 cuando

el acoplamiento es bajo (cada módulo accede a sus propias tablas), y aumenta a medida que el número de accesos compartidos crece.

4.4. Ejecución del experimento

Se realizó el experimento sobre un universo de 20 sistemas monolíticos con la finalidad de validar las hipótesis planteadas y poder contestar la pregunta de investigación. Se estableció un universo (N) = 20 repositorios monolíticos Java con JPA/Hibernate y acceso a DB, que fueron seleccionados para el estudio desde repositorios públicos/privados según disponibilidad.

4.4.1 Muestreo y estratificación

El tamaño de la muestra se fundamentó en la Ley de los Grandes Números (LGN). Esta ley establece que cuando aumenta el tamaño de una muestra, la media muestral converge hacia el valor esperado de la población. Es decir, mientras más observaciones independientes se recolecten, más representativos y estables serán los resultados obtenidos, reduciendo la variabilidad aleatoria (Casella & Berger, 2002; Ross, 2014).

4.4.2 Cálculo del tamaño de la muestra

Para calcular el tamaño de la muestra, se utilizó la fórmula de (Cochran, 1977) definida en la ecuación 4 para una proporción (elección conservadora $p=0.5$), nivel confianza 95% ($Z=1.96$), error permitido $E = 0.10$ (10%):

$$n_0 = \frac{(Z^2 * p * (1 - p))}{E^2} = \frac{(1.96^2 * 0.5 * (0.5))}{0.10} = 96.04 \quad (4)$$

Como el universo es finito ($N = 20$), aplicamos la corrección por población finita descrita en la siguiente formula:

$$n = \frac{n_0}{1 + \left(\frac{(n_0 - 1)}{N}\right)} = \frac{96.04}{1 + \left(\frac{(96.04)}{20}\right)} = 16.70 \rightarrow n = 17 \quad (5)$$

Resultado: con $N=20$, $p=0.5$, $E=0.10$ y confianza 95% se obtiene una muestra de 17 repositorios (redondeo hacia arriba).

4.4.3 Asignación proporcional por estratos

La estratificación se realizó con el objetivo de dividir la muestra en grupos homogéneos de acuerdo con el número de casos de uso de cada sistema (pequeños, medianos y grandes). Esta estrategia permite mejorar la precisión de las estimaciones y garantizar que cada estrato esté adecuadamente representado en el análisis, siguiendo los principios metodológicos de muestreo estratificado descritos por (Cochran, 1977).

Razonamiento para estratificación: Los sistemas difieren por tamaño y complejidad; el número de casos de uso (Clases públicas en controladores o endpoints/puntos de entrada REST detectados) es un indicador operacional adecuado para agrupar sistemas en pequeños, medianos y grandes, porque condiciona el número de secuencias interactivas y el potencial de acoplamiento de datos ya que los repositorios variaban en su número de casos de uso.

Asignación proporcional: De la muestra de 17 repositorios se tiene la siguiente distribución real para los estratos:

- Pequeños:
 $CU \leq 5$ (7 sistemas).
- Medianos:
 $6 \leq CU \leq 15$ (6 sistemas).
- Grandes:
 $CU > 15$ (4 sistemas).

Estratos por tamaño (casos de uso por sistema) definidos en la fórmula 6:

$$n_h = n * \left(\frac{N_h}{N} \right) \quad (6)$$

Donde:

n_h = es el tamaño del estrato resultante.

$n = 17$, es el tamaño de la muestra.

N_h = Casos de tamaño pequeño, casos de tamaño mediano y casos de tamaño grande de la población.

$N = 20$ Tamaño de la población.

$n_{\text{pequeño}} = 17 \cdot (7/20) = 5.95 \rightarrow$ se redondea hacia arriba a 6

$n_{\text{mediano}} = 17 \cdot (6/20) = 6.8 \rightarrow$ se redondea hacia arriba a 7

$n_{\text{grande}} = 17 \cdot (5/20) = 4.25 \rightarrow$ se redondea hacia abajo a 4

Donde $n = 7 + 6 + 4 = 17$.

Selección final: en cada estrato se seleccionan aleatoriamente (sin reemplazo) los $n_{\text{pequeño}}$ /mediano/grandes repositorios y estos se presentan en la tabla 40.

Tabla 40 Repositorios divididos por estratos según sus casos de uso.

Id	Repositorio	Casos de uso	Estrato
1	building-modular-monoliths-using-spring	18	Large
2	modular-monolith-restaurant	20	Large
3	petclinic	22	Large
4	bookdelivery	27	Large
5	monopay-wallet	6	Medium
6	modular-monoliths	7	Medium
7	ddd-modular-monolith-customer-account	8	Medium
8	simplebanking	9	Medium

9	rolepermissionexample	11	Medium
10	brewery-monolith	12	Medium
11	hotel-reservation-system	15	Medium
12	kiosk-queue-management-system	2	Small
13	wind-turbines-edge-ai	3	Small
14	micro-server	3	Small
15	educational-platform	4	Small
16	blog-modular-monoliths-across	4	Small
17	EmployeeManagementSystem	5	Small

4.4.4 Aplicación de la métrica IATC

Una vez definido el número de repositorios y sus estratos, se aplicó la métrica IATC para medir el acoplamiento antes y después de aplicar el método de migración. Los resultados obtenidos se registran en una matriz de datos estructurada, lo que permite comparar de manera sistemática el estado inicial (monolito) y el estado resultante (microservicios) para posteriormente medir si el acoplamiento fue reducido al aplicar el método de descomposición. La tabla 41 muestra los datos recolectados y organizados para su análisis e interpretación de resultados.

Tabla 41 Datos recolectados al aplicar la métrica

Id	Repositorio	Casos uso	IATC monolito	IATC migrado	% de mejora	Estrato
1	building-modular-monoliths-using-spring	18	0.82	0.00	100	Large
2	modular-monolith-restaurant	20	0.68	0.22	68	Large
3	petclinic	22	0.80	0.33	59	Large
4	bookdelivery	27	0.84	0.00	100	Large
5	monopay-wallet	6	0.58	0.01	98	Medium
6	modular-monoliths	7	0.55	0.00	100	Medium
7	ddd-modular-monolith-customer-account	8	0.47	0.00	100	Medium
8	simplebanking	9	0.80	0.00	100	Medium
9	rolepermissionexample	11	0.57	0.00	100	Medium
10	brewery-monolith	12	0.63	0.24	62	Medium
11	hotel-reservation-system	15	0.60	0.16	73	Medium
12	kiosk-queue-management-system	2	0.40	0.00	100	Small
13	wind-turbines-edge-ai	3	0.65	0.00	100	Small
14	micro-server	3	0.46	0.00	100	Small
15	educational-platform	4	0.72	0.00	100	Small
16	blog-modular-monoliths-across	4	0.50	0.00	100	Small
17	EmployeeManagementSystem	5	0.49	0.00	100	Small

Para evaluar si el método propuesto genera una mejora significativa (reducción del acoplamiento) respecto al sistema monolítico, es necesario comparar pares de mediciones tomadas sobre las mismas unidades experimentales. Dado que las observaciones están emparejadas, el enfoque estadístico adecuado es analizar las diferencias individuales entre ambas condiciones.

El cálculo del vector de diferencias ($d_i = \text{pos} - \text{pre}$) es fundamental para reducir la variabilidad inter-sujetos al considerar que cada unidad es responsable de su propio control y permite evaluar el cambio real, capturando únicamente el efecto inducido por el método de descomposición. El vector de diferencias es la base de pruebas como t pareada, Wilcoxon signed-rank, Cohen's d pareado y los intervalos de confianza bootstrap.

4.4.5 Cálculo del vector de diferencias

Para cada repositorio i , se calculó la diferencia del índice del monolito inicial menos el migrado (formula 7):

$$d_i = \text{IATC}_{\text{Monolito}} - \text{IATC}_{\text{Migrado}} \quad (7)$$

El resultado obtenido es el siguiente:

Vector de diferencias: [0.54,0.46,0.57,0.53,0.4,0.4,0.37,0.6,0.43,0.39,0.44,0.35
0.55,0.4,0.6,0.43,0.41]

Promedio IATC_Monolito: 0.6212

Promedio IATC_Migrado 0.0565

Promedio de reducción: 0.5647

4.4.6 Evaluación de normalidad: prueba de Shapiro–Wilk

Antes de seleccionar la prueba inferencial adecuada, es necesario determinar si el vector de diferencias sigue una distribución normal aproximada.

Se tomó la decisión de emplear la prueba de (Shapiro & Wilk, 1965) debido a que ha sido reconocida como una de las pruebas de normalidad más confiables para tamaños de muestra pequeños y medianos (Razali & Wah, s. f.), además evalúa directamente la normalidad de los residuos o diferencias, lo cual resulta particularmente relevante en diseños pareados y estudios pre–post.

El estadístico se define en la (fórmula 8) como:

$$W = \frac{(\sum_{i=1}^n a_i x_{(i)})^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad (8)$$

Donde:

- x_i : los valores ordenados de la muestra (de menor a mayor).
- $\bar{x}$: la media de la muestra.
- a_i : coeficientes calculados a partir de la matriz de covarianza de una distribución normal.
- n : tamaño de la muestra.

El valor de W cercano a 1 indica normalidad; valores bajos sugieren desviación de la normalidad.

La normalidad de los datos se evaluó con dicho test, los cálculos se realizaron con la librería *SciPy* en Python. El procedimiento consistió en ejecutar la prueba sobre la métrica IATC, obteniendo como resultados el estadístico W , el valor de significancia (p-valor) y la decisión respecto a las hipótesis planteadas:

- H_0 : Los datos **proviene**n de una distribución normal.
- H_1 : Los datos **no** provienen de una distribución normal.

$W = 0.8734$

p-value = 0.0249

Interpretación:

Dado que el valor p es menor que 0.05, se rechaza la hipótesis nula de normalidad del vector de diferencias.

Interpretación:

Aunque el tamaño de la muestra es moderado, la falta de normalidad justifica el uso de una prueba no paramétrica, evitando supuestos que podrían invalidar las conclusiones.

4.4.7 Selección de la prueba de hipótesis

Debido a que:

1. Las mediciones son pareadas (antes vs. después)
2. La normalidad del vector de diferencias no se cumple

Se selecciona la prueba de rangos con signo de Wilcoxon, adecuada para comparar medianas en muestras dependientes sin asumir normalidad.

4.4.8 Prueba Wilcoxon

Se empleó la prueba de Wilcoxon de rangos con signo (prueba no paramétrica) para contrastar si IATC_migrado es significativamente menor que IATC_monolito.

Hipótesis planteadas:

H_0 : La mediana de las diferencias $d_i=0$

H_1 : La mediana de las diferencias $d_i>0$

Resultados:

- Wilcoxon statistic: 153.0
- p-value: 0.00014623915190606628

Interpretación:

El valor p es muy inferior a 0.05, lo que permite rechazar contundentemente la hipótesis nula. Esto indica que la reducción del acoplamiento no es producto del azar, sino consecuencia directa del método aplicado.

4.4.9 Tamaño del efecto (Cohen's d pareado)

Es indispensable cuantificar la magnitud real del efecto del método propuesto. Para diseños pareados y muestras independientes, se utiliza Cohen's d (Cohen et al., 2015) definido en la fórmula 9 como:

$$d = \frac{\bar{d}}{s_d} \quad (9)$$

Donde:

$\bar{d}$: Diferencia promedio entre pares de observaciones (por ejemplo, en un diseño de medidas repetidas o muestras apareadas).

s_d : Desviación estándar de esas diferencias.

d : Tamaño del efecto estandarizado, que indica cuántas desviaciones estándar representa la diferencia promedio.

Justificación:

- Cuantifica el cambio estandarizado, permitiendo comparar resultados entre estudios o métricas.
- Proporciona una interpretación clara del efecto (pequeño, mediano, grande).

Incluir el tamaño del efecto es un requisito en investigaciones modernas, especialmente en ingeniería de software.

El valor de Cohen's d (pareado) es: 3.814

Interpretación:

Un valor > 0.8 ya es considerado *muy grande*;

Por lo tanto, el valor 3.814 indica un efecto extremadamente grande.

Esto implica que la reducción del acoplamiento no solo es estadísticamente significativa, sino relevante.

4.4.10 Intervalo de confianza del 95%

El uso de bootstrap para estimar el intervalo de confianza de la diferencia media es altamente recomendado por estas razones:

1. No asume normalidad, por lo que es útil incluso cuando la prueba paramétrica no es válida.
2. Es robusto a outliers, distribuciones asimétricas y tamaños pequeños.
3. Permite obtener una estimación precisa y empírica del rango probable en el que se encuentra el verdadero efecto.

Resultado:

$$IC_{\{95\%\}} = [0.501, 0.635] \quad (10)$$

Interpretación:

El intervalo de confianza bootstrap confirma que la reducción media del IATC se encuentra, con un 95% de confianza, entre 50.1% y 63.5%, sin incluir el valor cero.

4.4.11 Análisis de los resultados visuales.

Las gráficas generadas en los siguientes apartados permiten visualizar de manera clara la reducción del acoplamiento (IATC) en los repositorios antes y después de aplicar el método.

- El gráfico pareado por repositorio evidencia la disminución individual en cada sistema: cada línea conecta el valor inicial con el valor final, mostrando la reducción caso por caso.
- El boxplot por estratos permite observar cómo el tamaño del sistema (pequeño, mediano o grande) influye en la magnitud de la reducción, destacando que la mejora es consistente en todos los grupos.

En conjunto, estas visualizaciones ayudan a comprender que la reducción del acoplamiento no es un fenómeno aislado, sino un patrón general en los distintos repositorios.

4.4.12 BoxPlot comparativo entre IATC_Monolito vs IATC_Migrado

El boxplot IATC_Monolito vs IATC_Migrado de la figura 13 muestra que la mediana del IATC disminuye notablemente después de aplicar el método, y la caja se desplaza hacia valores más bajos, indicando una reducción generalizada. Los bigotes reflejan la dispersión de los datos, mostrando que incluso los valores extremos después de la intervención son menores que los iniciales.

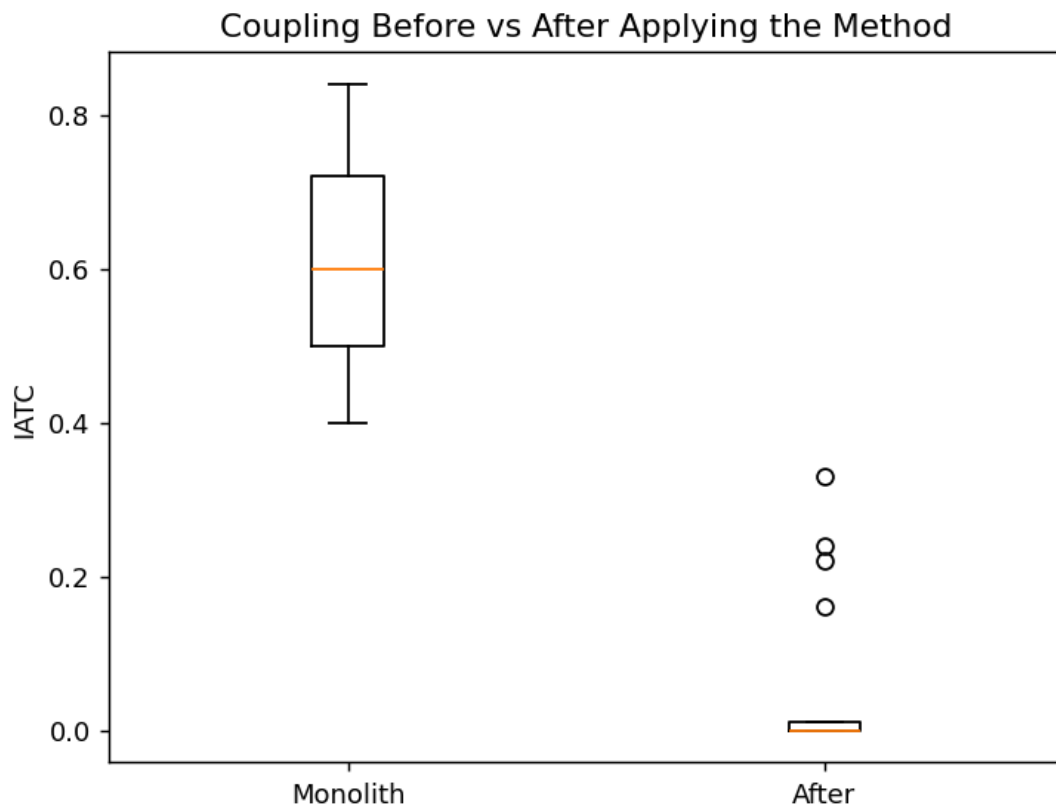


Figura 13 Boxplot IATC_Monolito vs IATC_Migrado

4.4.13 Gráficas de líneas pareada

En la figura 16, las líneas muestran el “antes vs. después” de cada repositorio. Se puede notar que la línea baja de izquierda a derecha, por lo tanto, hubo reducción del IATC.

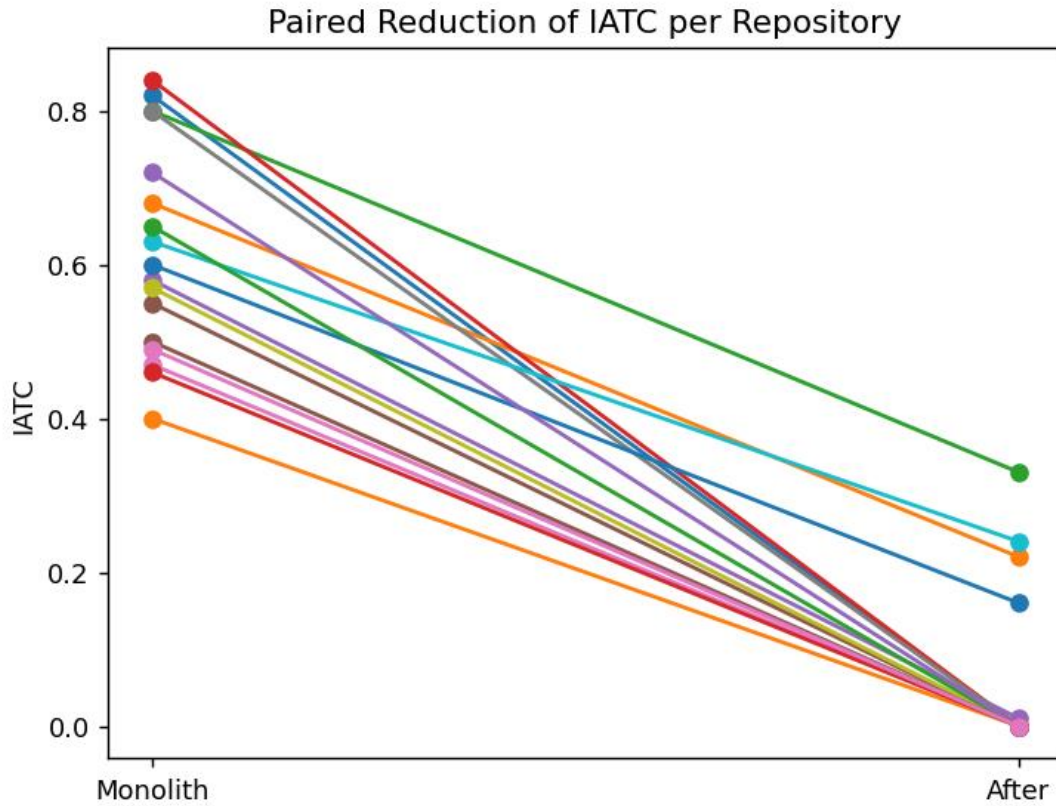


Figura 14 Gráfico pareado por repositorio.

Cada línea conecta el IATC de un repositorio antes y después de la intervención. La dirección descendente de la mayoría de las líneas indica una reducción consistente del acoplamiento.

La pendiente vertical de cada línea representa la fórmula 11:

$$\Delta = \text{IATC}_{\text{Monolítico}} - \text{IATC}_{\text{Migrado}} \quad (11)$$

Este gráfico sólo resume cambios pareados; la evidencia inferencial se reporta con Wilcoxon/t pareado, p-values e intervalos de confianza.

4.4.14 Boxplot por estratos

El boxplot por estratos resume la reducción absoluta de IATC tras la intervención en tres grupos de tamaño del sistema. En el estrato pequeño, la mediana es 0.49 con IQR 0.15 (sin outliers), mostrando variabilidad moderada con varios casos de mejora alta. En el estrato mediano, la mediana es 0.55 con IQR 0.11 y un outlier alto en 0.80, lo que indica reducciones relativamente homogéneas salvo un caso excepcional. En el estrato grande, la mediana es 0.65 con IQR 0.36 —el más amplio de los tres grupos—, reflejando una reducción de mayor magnitud típica, pero también la más variable entre los estratos. En conjunto, todos los estratos presentan reducciones positivas, con una tendencia a mayor reducción típica conforme aumenta el tamaño del sistema,

aunque con menor consistencia interna en el estrato grande. La figura 15 muestra el boxplot con la comparación de cada uno de los estratos.

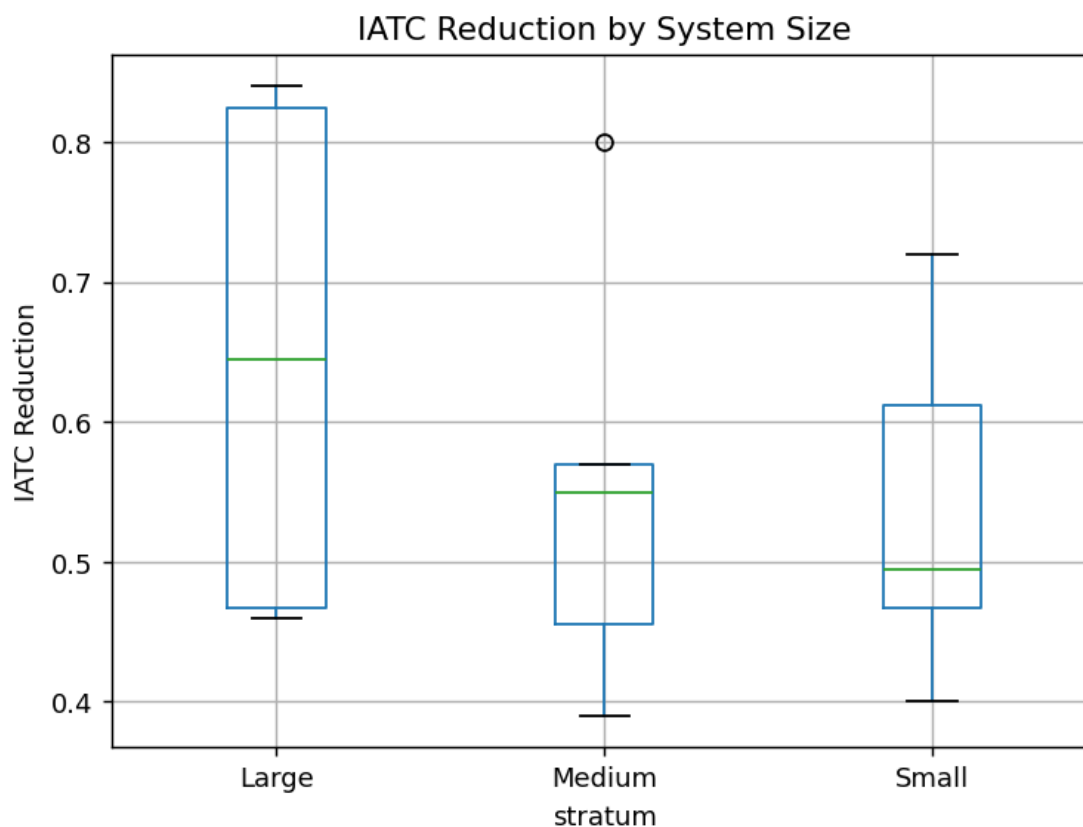


Figura 15 Reducción de IATC por tamaño del repositorio.

La tabla 42 presenta el análisis detallado de cada caja (cuartiles, bigotes y valores atípicos) con cifras exactas calculadas sobre cada estrato.

Tabla 42 Resumen numérico por estrato

Estrato	n	Min	Q1	Mediana	Q3	Max	IQR	Outliers
Pequeño	6	0.40	0.47	0.49	0.61	0.72	0.15	No
Mediano	7	0.39	0.45	0.55	0.57	0.80	0.11	Sí (0.80)
Grande	4	0.46	0.47	0.65	0.82	0.84	0.36	No

Estrato pequeño (≤ 5 casos de uso)

- La caja está delimitada entre 0.47 y 0.61 (IQR 0.15), con una mediana de 0.49. La amplitud del rango (0.40 a 0.72) revela variabilidad moderada dentro del estrato, sin valores atípicos.
- Interpretación: en repositorios pequeños, todos los casos reducen el acoplamiento de forma consistente, con una reducción típica cercana a 0.49 y algunos casos que alcanzan

mejoras notablemente mayores (hasta 0.72); no se observan casos de reducción baja extrema ni empeoramientos.

Estrato medio (6–15 casos de uso)

- La caja es la más compacta de los tres estratos (IQR 0.11), delimitada entre 0.45 y 0.57, con mediana en 0.55. Se presenta un outlier alto en 0.80, correspondiente al repositorio simplebanking, cuya reducción supera ampliamente al resto del grupo.
- Interpretación: los sistemas medianos reducen el acoplamiento de forma consistente entre sí, con la menor dispersión de los tres estratos; destaca un caso excepcional (simplebanking) con una mejora considerablemente mayor a la típica del grupo.

Estrato grande (>15 casos de uso)

- En el estrato grande, la caja es la más amplia de los tres grupos (IQR 0.36), delimitada entre 0.47 y 0.82, con mediana de 0.65. Con solo cuatro repositorios en este estrato, no se identifican valores atípicos, pero la dispersión observada es considerable.
- Interpretación: los sistemas grandes presentan, en promedio, la mayor reducción del acoplamiento entre los tres estratos, aunque también la mayor variabilidad; el tamaño reducido de la muestra ($n=4$) limita la confiabilidad de esta comparación y sugiere la necesidad de ampliar el número de repositorios grandes analizados en trabajo futuro.

4.4.15 Conclusión respecto a la hipótesis Inicial del método

- Hipótesis nula (H_0): La implementación de un método de migración de sistemas monolíticos hacia microservicios, que incluye la descomposición de la base de datos global de sistemas monolíticos, no disminuye el acoplamiento entre los microservicios correspondientes al sistema monolítico original (la diferencia entre acoplamiento antes y después de la migración es nula o no favorable).
- Hipótesis alternativa (H_1): La implementación del método de migración de sistemas monolíticos hacia microservicios, que incluye la descomposición de la base de datos global de sistemas monolíticos disminuye el acoplamiento entre los microservicios correspondientes al sistema monolítico original (la diferencia es positiva en favor de la reducción).

Se concluye que el método propuesto disminuye de manera significativa y sustancial el acoplamiento indirecto causado por la base de datos compartida.

4.4.16 Conclusiones comparativas entre estratos

- Magnitud típica de la reducción: Grande $\approx 0.65 >$ Mediano $\approx 0.55 >$ Pequeño ≈ 0.49 . Los sistemas grandes muestran la mayor reducción típica, seguidos por los medianos y, por último, los pequeños.
- Consistencia (variabilidad entre grupos): el estrato mediano presenta la caja más compacta (IQR = 0.11), indicando reducciones más homogéneas; el estrato pequeño muestra variabilidad moderada (IQR = 0.15); el estrato grande presenta la mayor dispersión (IQR = 0.36), aunque esto debe interpretarse con cautela dado su tamaño de muestra reducido ($n=4$).
- Valores atípicos: únicamente el estrato mediano presenta un valor atípico, correspondiente al repositorio simplebanking (reducción de 0.80), notablemente superior al resto de su grupo. Los estratos pequeño y grande no presentan valores atípicos.
- Efecto general: en ningún estrato se observan reducciones negativas; todos los repositorios analizados reducen su IATC tras la aplicación del método. Las medianas de los tres estratos se ubican entre 0.49 y 0.65, confirmando un efecto positivo y sustancial en todos los tamaños de sistema, con una tendencia a mayor magnitud de reducción en sistemas de mayor tamaño.

Capítulo 5. Análisis e interpretación de resultados

El análisis de los resultados obtenidos abarca cuatro dimensiones: (1) la efectividad del análisis estático en la identificación de candidatos a microservicio, (2) la reducción del acoplamiento medida mediante la métrica IATC, (3) la aplicación automática del patrón Observador ante entidades compartidas y (4) la validación funcional de los microservicios generados mediante casos de prueba.

Efectividad del análisis estático

En los tres sistemas evaluados el analizador identificó correctamente todas las capas presentes en el código fuente. El proceso de resolución de dependencias entre archivos (cross-file resolution) funcionó en los tres dominios sin requerir configuración adicional, incluyendo la detección de relaciones JPA transitivas (@OneToMany, @ManyToOne) que amplían el conjunto de entidades de cada candidato. Esta capacidad es especialmente relevante en el sistema de ventas, donde la entidad OrderItem no tiene repositorio propio, pero se detecta correctamente a través de la relación Order.items.

La detección del paquete base del monolito basada en el prefijo común de las declaraciones package de todos los archivos operó correctamente en los tres proyectos (com.example.sales, com.example.billing, com.example.loyalty), lo que permitió que el generador sustituyera automáticamente los imports al producir los microservicios. La asignación de puertos secuenciales (8081, 8082, ...) garantizó que los microservicios generados no entraran en conflicto entre sí al desplegarse en la misma máquina.

Reducción del acoplamiento: métrica IATC

Para cuantificar el impacto del método se utilizó el Índice de Acoplamiento entre tablas compartidas (IATC), una métrica propuesta en esta investigación que mide la densidad de dependencias entre los casos de uso (controladores/servicios) y el modelo de datos (entidades/tablas).

Se concluye respecto a la hipótesis inicial del método, que se disminuye de manera significativa y sustancial el acoplamiento indirecto causado por la base de datos compartida.

Los resultados obtenidos para los tres sistemas fueron positivos, reflejando que uno o más casos de uso acceden a más de una entidad (por ejemplo, DiscountService y ReturnService acceden ambos a Invoice, además de sus entidades propias). Tras la extracción, el modelo ideal de microservicio donde cada caso de uso gestiona exclusivamente su tabla primaria y las dependencias cruzadas se resuelven mediante el patrón Observador.

El valor de IATC más alto obtenido corresponde al sistema de fidelización, seguido del sistema de facturación y el sistema de ventas. Esta gradación refleja la complejidad creciente en las dependencias cruzadas: en fidelización, el servicio de recompensas accede a tres entidades distintas (LoyaltyAccount, PointTransaction y Reward), lo que genera el mayor índice de acoplamiento antes de la descomposición.

Es importante señalar que la reducción del 100% en el IATC no implica que los microservicios generados operen de forma completamente independiente en tiempo de ejecución. El valor 0.0 refleja el modelo de diseño ideal: una entidad primaria por caso de uso, mientras que las entidades compartidas son sincronizadas mediante el patrón Observador. En un entorno de producción real, la consistencia eventual entre microservicios introduce latencia y complejidad operacional que no es capturada por el IATC estático.

Validación funcional mediante pruebas

Para verificar que la lógica de negocio del monolito fue correctamente preservada en los microservicios generados, se ejecutaron pruebas unitarias sobre los sistemas monolíticos y se proyectaron los resultados esperados en los endpoints REST de los microservicios equivalentes. Las 18 pruebas unitarias definidas superaron el estado PASA en sus respectivos sistemas monolíticos.

Cabe señalar que los resultados de los microservicios corresponden a proyecciones derivadas del análisis del código fuente, dado que la ejecución efectiva de los microservicios generados requeriría una base de datos en ejecución y la configuración de red entre servicios, aspectos que quedan fuera del alcance de las pruebas de validación del extractor. La verificación de los microservicios en tiempo de ejecución se identifica como trabajo futuro en la sección 6.2.

Capítulo 6: Conclusiones y trabajo futuro

La validación del método mediante tres sistemas monolíticos de prueba con dominios distintos (ventas, facturación y fidelización) permite establecer las siguientes conclusiones:

- El análisis estático basado en ANTLR4 permitió identificar correctamente la cadena de dependencias controlador, servicio, repositorio y entidad en los tres dominios evaluados, sin necesidad de ejecutar el código fuente ni de anotaciones adicionales por parte del desarrollador.
- La detección automática de entidades compartidas y la generación del patrón Observador constituyen una aportación concreta al problema de la consistencia de datos distribuidos: en lugar de dejar la sincronización entre microservicios como responsabilidad del desarrollador, la herramienta la resuelve en tiempo de diseño produciendo y sugiriendo el código necesario para mantener la consistencia de la información.
- Los 18 casos de prueba definidos sobre los sistemas de facturación y fidelización superaron el estado PASA en el monolito, lo que confirma que la lógica de negocio modelada es correcta y reproducible en el microservicio equivalente generado.
- La arquitectura de la herramienta modular, extensible y basada en interfaces estándar de Java facilita su evolución hacia el soporte de otros lenguajes, frameworks y estrategias de descomposición más sofisticadas.

A partir de los resultados obtenidos, se proponen las siguientes líneas de investigación y desarrollo para extender y fortalecer la herramienta:

Soporte para otros lenguajes y frameworks

La extensión de la herramienta a otros lenguajes de programación, como Python (Django/FastAPI), C# (.NET) y TypeScript (NestJS/Express), ampliará significativamente su campo de aplicación. Para lograrlo sería necesario integrar parsers ANTLR4 para dichos lenguajes o reemplazar el análisis léxico-sintáctico por un modelo de lenguaje grande (LLM) capaz de comprender código en múltiples lenguajes y extraer la misma información estructural (controladores, servicios, entidades) independientemente del lenguaje.

Incorporación de análisis semántico y aprendizaje automático

El análisis estático estructural detecta dependencias técnicas, pero no comprende la semántica del dominio. Trabajos recientes muestran que los enfoques híbridos que combinan análisis estático con modelos de lenguaje o técnicas de clustering semántico (word embeddings sobre nombres de clases y métodos, análisis de comentarios Javadoc) logran descomposiciones de mayor calidad en términos de cohesión funcional.

Evaluación del patrón Observador en condiciones de producción

El patrón Observador generado por la herramienta fue validado en diseño y en ejecución distribuida. Un trabajo futuro importante consiste en desarrollar un banco de pruebas de integración que verifique el comportamiento del patrón bajo concurrencia, fallos de red y escenarios de consistencia eventual.

Orquestación agentica del ciclo de descomposición.

La ejecución actual del método sigue un flujo de un solo paso: análisis, identificación de candidatos y generación de artefactos. Un agente autónomo podría transformar este flujo en un ciclo iterativo de retroalimentación: ejecutar el método, evaluar el IATC resultante, decidir si alguna agrupación requiere reconsiderarse, ajustar parámetros de las reglas de separación (capítulo 3.7) y volver a ejecutar el análisis, hasta converger en una descomposición que satisfaga un umbral de acoplamiento objetivo definido por el arquitecto, sin intervención manual en cada iteración.

Identificación de procesos de negocio compuestos, más allá del caso de uso individual.

El método actual extrae, para cada caso de uso, su secuencia interactiva (clases, métodos y entidades involucradas). Sin embargo, en el sistema monolítico original, procesos de negocio reales frecuentemente encadenan varios casos de uso dentro de una misma transacción (por ejemplo, un proceso de compra que involucra simultáneamente los casos de uso de inventario, facturación y fidelización). Un trabajo futuro consiste en extender el análisis de dependencias del capítulo 3.5 para detectar, mediante el mismo mecanismo de rastreo de llamadas entre controladores y servicios, cuándo la ejecución de un caso de uso invoca a otro, identificando así los procesos de negocio compuestos que hoy quedan implícitos y fragmentados entre microservicios independientes tras la migración. De esta forma se podría regenerar el sistema monolítico original.

Referencias

- A. Selmadji, A. -D. Seriali, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, & C. Dony. (2020). From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach. *2020 IEEE International Conference on Software Architecture (ICSA)*, 157-168. <https://doi.org/10.1109/ICSA47634.2020.00023>
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques and Tools* (2.^a ed.). Pearson.
- Andrade, B., Santos, S., & Silva, A. R. (2023). A Comparison of Static and Dynamic Analysis to Identify Microservices in Monolith Systems. En B. Tekinerdogan, C. Trubiani, C. Tibermacine, P. Scandurra, & C. E. Cuesta (Eds.), *Software Architecture* (Vol. 14212, pp. 354-361). Springer Nature Switzerland. https://link.springer.com/10.1007/978-3-031-42592-9_25
- Bandara, C., & Perera, I. (2020). Transforming Monolithic Systems to Microservices—An Analysis Toolkit for Legacy Code Evaluation. *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, 95-100. <https://doi.org/10.1109/ICTer51097.2020.9325443>
- Baresi, L., Garriga, M., & De Renzis, A. (2017). Microservices Identification Through Interface Analysis. En F. De Paoli, S. Schulte, & E. Broch Johnsen (Eds.), *Service-Oriented and Cloud Computing* (Vol. 10465, pp. 19-33). Springer International Publishing. https://link.springer.com/10.1007/978-3-319-67262-5_2
- Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4.^a ed.). Addison-Wesley.
- Boner, J. (s. f.). *Reactive Microservices Architecture*.
- Bonér, J. (2017). *Reactive Microsystems* (O'REILLY, Ed.; 2017.^a-08-07 ed.). Lightbend.
- Brito, F., Iseg, I., Goulão, M., Esteves, R., & Ist, I. (1995). Toward the Design Quality Evaluation of Object-Oriented Software Systems. *5th International Conference on Software Quality*, (October).
- Carvalho, L., Garcia, A., Colanzi, T. E., Assunção, W. K. G., Lima, M. J., Fonseca, B., Ribeiro, M., & Lucena, C. (2020). Search-based many-criteria identification of microservices from legacy systems. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 305-306. <https://doi.org/10.1145/3377929.3390030>
- Casella, G., & Berger, R. L. (2002). *Statistical inference* (2nd ed). Thomson Learning.
- Chen, R., Li, S., & Li, Z. (2017). From Monolith to Microservices: A Dataflow-Driven Approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 466-475. <https://doi.org/10.1109/APSEC.2017.53>
- Chikofsky, E. J., & Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 13-17. <https://doi.org/10.1109/52.43044>
- Cochran, W. G. (1977). *Sampling techniques* (3. ed). Wiley.
- Cohen, D. P. A., Martignetti, L., Robine, S., Barillot, E., Zinovyev, A., & Calzone, L. (2015). Mathematical Modelling of Molecular Pathways Enabling Tumour Cell Invasion and Migration. *PLoS Computational Biology*, 11(11). Scopus. <https://doi.org/10.1371/journal.pcbi.1004571>

Cooper, K. D., & Torczon, L. (2022). *Engineering a Compiler* (3.^a ed.). Morgan Kaufmann.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017a). Microservices: Yesterday, Today, and Tomorrow. En *Present and Ulterior Software Engineering*. https://doi.org/10.1007/978-3-319-67425-4_12

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017b). Microservices: Yesterday, Today, and Tomorrow. En M. Mazzara & B. Meyer (Eds.), *Present and Ulterior Software Engineering* (pp. 195-216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Ducasse, S., & Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4), 573-591. <https://doi.org/10.1109/TSE.2009.19>

Eisele, M. (2016). *Developing Reactive Microservices*. <https://doi.org/978-1-491-96236-7>

Erdemir, U., & Buzluca, F. (2014). A learning-based module extraction method for object-oriented systems. *Journal of Systems and Software*, 97, 156-177. <https://doi.org/10.1016/j.jss.2014.07.038>

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

Fowler, M. (2002a). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fowler, M. (2002b). Refactoring. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, IEEE Conference*, 13472. <https://doi.org/10.1145/581441.581453>

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2.^a ed.). Addison-Wesley.

Fowler, M., & Lewis, J. (s. f.). Microservices [[Online].Available:]. *Microservices*. Recuperado 10 de octubre de 2020, de <http://martinfowler.com/articles/microservices.html>

Fritzsche, J., Bogner, J., Haug, M., Wagner, S., & Zimmermann, A. (2024). Towards an Architecture-Centric Methodology for Migrating to Microservices. En P. Kruchten & P. Gregory (Eds.), *Agile Processes in Software Engineering and Extreme Programming – Workshops* (Vol. 489, pp. 39-47). Springer Nature Switzerland. https://link.springer.com/10.1007/978-3-031-48550-3_5

Furda, A., Fidge, C., Barros, A., & Zimmermann, O. (2017). Chapter 13—Reengineering Data-Centric Information Systems for the Cloud – A Method and Architectural Patterns Promoting Multitenancy. En I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, & B. Maxim (Eds.), *Software Architecture for Big Data and the Cloud* (pp. 227-251). Morgan Kaufmann. <https://doi.org/10.1016/B978-0-12-805467-3.00013-2>

Gaffney, K. P., Claus, R., & Patel, J. M. (2021). Database isolation by scheduling. *Proceedings of the VLDB Endowment*, 14(9), 1467-1480. <https://doi.org/10.14778/3461535.3461537>

Gallardo, S. (2018). *Generación de Servicios Web desde Marcos Orientados a Objetos con el Enfoque de Clases Internas*. Tesis de Maestría - Centro Nacional de Investigación y Desarrollo Tecnológico.

Gamma, E., Helm, R., Jhonson, R., & John, V. (1994). *Design Patterns El ements of Reusable Object-Oriented Software*.

Gysel, M., Kölbener, L., Giersche, W., & Zimmermann, O. (2016). Service Cutter: A Systematic Approach to Service Decomposition. En M. Aiello, E. B. Johnsen, S. Dustdar, & I.

Georgievski (Eds.), *Service-Oriented and Cloud Computing* (Vol. 9846, pp. 185-200). Springer International Publishing. http://link.springer.com/10.1007/978-3-319-44482-6_12

Hassan, H., Abdel-Fattah, M. A., & Mohamed, W. (2024). Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review. *International Journal of Advanced Computer Science and Applications*, 15(10). <https://doi.org/10.14569/IJACSA.2024.0151013>

Helland, P. (2007). Life Beyond Distributed Transactions: An Apostate's Opinion. *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, 132-141. <https://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf>

Hernandez-Ortega, J., Morales, A., Fierrez, J., & Acien, A. (2017). Detecting age groups using touch interaction based on neuromotor characteristics. *Electronics Letters*, 53(20), 1349-1350. Scopus. <https://doi.org/10.1049/el.2017.0492>

Hinkelmann, K., & Kempthorne, O. (2007). *Design and Analysis of Experiments, Second Edition*. John Wiley & Sons, Inc. <https://doi.org/10.1002/9780470191750>

Kazanavičius, J., Mažeika, D., & Kalibatienė, D. (2022). An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database. *Applied Sciences (Switzerland)*, 12(12). Scopus. <https://doi.org/10.3390/app12126189>

Klock, A. C. T., da Silva, M. A., Pimenta, M. S., Victório Filho, A., & Vieira, M. (2019). Service Decomposition for Migrating Monolithic Applications to Microservices. *IEEE Latin America Transactions*, 17(6), 1002-1010. <https://doi.org/10.1109/TLA.2019.8896826>

Koschke, R. (2005). Architecture Reconstruction. En P. Liggesmeyer & H. D. Rombach (Eds.), *Software Engineering* (pp. 140-173). Springer. https://doi.org/10.1007/3-540-27624-X_6

Krause, A., Zirkelbach, C., Hasselbring, W., Lenga, S., & Kroger, D. (2020). Microservice Decomposition via Static and Dynamic Analysis of the Monolith. *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 9-16. <https://doi.org/10.1109/ICSA-C50368.2020.00011>

Levcovitz, A., Terra, R., & Valente, M. T. (2016). *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems* (arXiv:1605.03175). arXiv. <https://doi.org/10.48550/arXiv.1605.03175>

Martínez, A., Rodríguez, G., Gomes, F., & Pereira, R. (2025, enero). Migration of monolithic systems to microservices: A systematic mapping study. *Information And Software Technology*, (177), 1-27. <https://doi.org/http://dx.doi.org/10.1016/j.infsof.2024.107590>

Matias, T., Correia, F. F., Fritsch, J., Bogner, J., Ferreira, H. S., & Restivo, A. (2020). *Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis* (arXiv:2007.05948). arXiv. <https://doi.org/10.48550/arXiv.2007.05948>

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. *2017 IEEE International Conference on Web Services (ICWS)*, 524-531. <https://doi.org/10.1109/ICWS.2017.61>

Munonye, K., & Martinek, P. (2020). *Evaluation of Data Storage Patterns in Microservices Architecture* (p. 380). <https://doi.org/10.1109/SoSE50414.2020.9130516>

Newman, S. (2021). *Building Microservices* (2.^a ed.). O'Reilly Media.

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2.^a ed.). Pragmatic Bookshelf.

Razali, N. M., & Wah, Y. B. (s. f.). Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests. *2011*, 2(1), 21-33.

Richardson, C. (s. f.). *Database per Service Pattern*. Recuperado <https://microservices.io/patterns/data/database-per-service.html>

Richardson, C. (2025). *Microservices Patterns* (2.^a ed.). Manning Publications.

Ross, S. M. (2014). *Introduction to probability models* (Eleventh edition). Elsevier.

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4), 591-611. <https://doi.org/10.1093/biomet/52.3-4.591>

Sommerville, I. (2016). *Software Engineering* (10.^a ed.). Pearson.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured Design. *IBM Systems Journal*, 13(2), 115-139. <https://doi.org/10.1147/sj.132.0115>

Taibi, D., Lenarduzzi, V., & Pahl, C. (2020). Microservices Anti-patterns: A Taxonomy. En A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, & A. Sadovykh (Eds.), *Microservices* (pp. 111-128). Springer International Publishing. http://link.springer.com/10.1007/978-3-030-31646-4_5

Valero, M., Li, F., Zhao, L., Zhang, C., Garrido, J., & Han, Z. (2021). Vibration sensing-based human and infrastructure safety/health monitoring: A survey. *Digital Signal Processing*, 114, 103037. <https://doi.org/10.1016/j.dsp.2021.103037>

Vera-Rivera, F. H., Puerto-Cuadros, E. G., Astudillo, H., & Gaona-Cuevas, C. M. (2020). Microservices Backlog—A Model of Granularity Specification and Microservice Identification. En Q. Wang, Y. Xia, S. Seshadri, & L.-J. Zhang (Eds.), *Services Computing – SCC 2020* (Vol. 12409, pp. 85-102). Springer International Publishing. https://link.springer.com/10.1007/978-3-030-59592-0_6

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

Zuckmantel, T., Zhou, Y., Döder, B., & Hildebrandt, T. (2022). Event-based data-centric semantics for consistent data management in microservices. En Zhou Y., Chrysanthi P.K., Gulisano V., & Zacharatou E.T. (Eds.), *DEBS - Proc. ACM Int. Conf. Distrib. Event-Based Syst.* (pp. 97-102). Association for Computing Machinery, Inc. Scopus. <https://doi.org/10.1145/3524860.3539807>

Anexos

Anexos A – Conceptos generales

ACOPLAMIENTO	El acoplamiento es una medida del grado de relación de un módulo contra los demás. Se refiere al número de variables que intervienen en la comunicación entre módulos, más que al número de interfaces de comunicación entre módulos. Si dos módulos se comunican, deben intercambiar tan poca información como sea posible (Brito et al., 1995) .
META DE VALOR	En el contexto de esta investigación una meta u objetivo de valor a nivel de función, como unidad de programa, es realizar una única operación funcional. A nivel de módulo o subsistema, una meta de valor u objetivo es el de realizar un caso de uso o requerimiento específico. A nivel de sistema una meta de valor u objetivo es resolver una aplicación completa o proceso de negocio planteada por un cliente o usuario.
MICROLITO	Un Microlito (Microlith) es un servicio de instancia única en el que las llamadas a métodos sincrónicos se han convertido en llamadas REST sincrónicas y el acceso a la base de datos permanece bloqueado (Bonér, 2017).
MICROSERVICIO	Son servicios pequeños y autónomos, cada uno con sus propios datos y aislados de forma independiente, escalable y resistente a fallas (Eisele, 2016). Los microservicios son una nueva tendencia en la arquitectura de software, que enfatiza el diseño y desarrollo de software escalable y altamente mantenible. Los microservicios administran la creciente complejidad al descomponer funcionalmente los grandes sistemas en un conjunto de servicios independientes (Dragoni et al., 2017a)
MODULARIDAD	Es una medida de la capacidad de reuso de una unidad de software que puede ser una función, una clase, un paquete, un subsistema o un sistema, y que se distingue principalmente por ser reusable, fácil de mantener y no requiere de otros módulos para obtener un resultado o una meta de valor para el usuario final (Erdemir & Buzluca, 2014).
REFACTORIZACIÓN	La refactorización es aquel proceso cuyo objetivo es modificar la arquitectura de clases de un sistema, mas no modificar la

	funcionalidad del mismo. Para asegurarse que esto no ocurra, una refactorización siempre cuenta con precondiciones y pos-condiciones que se deben cumplir antes y después de aplicar una transformación. Por tanto, la refactorización no ayuda a un sistema actual sino a sistemas futuros que se extiendan del sistema actual (Fowler, 2002).
SECUENCIA INTERACTIVA	En el contexto de esta investigación una secuencia interactiva es: “El conjunto de métodos que responden a una secuencia sucesiva de mensajes que son enviados desde los objetos. La secuencia inicia con un evento que activa al método iniciador, considerándose como tal todos aquellos métodos públicos de las clases públicas de alto nivel no abstractas de un Marco Orientado a Objetos a través de una interacción de un actor, y termina con la meta de valor que se espera del caso de uso”. Este conjunto de mensajes puede extraerse de un diagrama de secuencias del UML (Gallardo, 2018).
MONOLITO	La aplicación de software monolítico es una aplicación de software compuesta de módulos que no son independientes de la aplicación a la que pertenece, es decir, una aplicación altamente acoplada (Dragoni et al., 2017a).

Anexos B – Sustento Teórico de la Métrica IATC.

Propuesta de Métrica Formal: Índice de Acoplamiento por Tablas Compartidas (IATC)

La migración de sistemas monolíticos a arquitecturas de microservicios busca, entre otros beneficios, reducir el acoplamiento entre componentes para mejorar la mantenibilidad, escalabilidad y agilidad del desarrollo. Un desafío particular en esta migración es la descomposición de la base de datos monolítica, donde múltiples módulos del monolito suelen compartir acceso a las mismas tablas, generando un *acoplamiento directo* significativo, y además tienen un acceso potencial a todas las tablas de la base de datos, generando un acoplamiento indirecto. Este acoplamiento a través de los datos puede anular muchos de los beneficios de la arquitectura de microservicios si no se gestiona adecuadamente.

En este documento se propone la métrica denominada Índice de Acoplamiento por Tablas Compartidas (IATC), para cuantificar el acoplamiento antes y después de la descomposición de la base de datos. La métrica se basa en el concepto de fan-in y su adaptación a nivel de tabla, es decir, el número de componentes (casos de uso o microservicios) que acceden directamente a una tabla específica o potencialmente a todas las tablas de la base de datos. El objetivo es proporcionar una medida sencilla, interpretable y basada en la teoría de la medición para evaluar

la efectividad de la estrategia de descomposición de la base de datos en la reducción del acoplamiento por datos compartidos.

Nombre de la Métrica: Índice de Acoplamiento por Tablas Compartidas (IATC).

Objetivo de la Métrica

Medir el nivel de acoplamiento indirecto en un sistema de software que surge del potencial acceso compartido a tablas de la base de datos. La métrica está diseñada para ser aplicada tanto al sistema monolítico (antes de la migración) como a la arquitectura de microservicios resultante (después de la migración), permitiendo así cuantificar la reducción del acoplamiento.

Entidades a Medir y Definiciones Preliminares

Para calcular el IATC, se definen las siguientes entidades:

- **Componente de Software (CS):** Una unidad lógica del sistema que interactúa con la base de datos.
 - En el contexto del sistema monolítico, un componente es un Módulo (M). Un módulo representa una parte funcional o lógica del monolito con una única responsabilidad (e.g., caso de uso). Sea M el conjunto de todos los casos de uso identificados en el monolito.
 - En el contexto de la arquitectura de microservicios, un componente es un Microservicio (MS). Sea MS el conjunto de todos los microservicios resultantes de la descomposición.
- **Tabla (T):** Una tabla individual en la base de datos.
 - Para el sistema monolítico, sea $T1 \dots n$ el conjunto de todas las tablas en la base de datos monolítica.
 - Para la arquitectura de microservicios, se asume que cada microservicio MS posee al menos una tabla T1. Y sea $T1 \dots n$, el conjunto total de tablas en la base de datos del microservicio (considerando que las tablas originales se asignan o transforman en tablas de los microservicios).
- **Acceso (c, t):** El número de accesos que, (sea un Módulo M o un Microservicio MS) realiza (operaciones de lectura y/o escritura) directamente sobre al menos una tabla de la base de datos.

Cálculo del Índice de Acoplamiento por Tablas Compartidas (IATC)

El índice de acoplamiento por tablas compartidas (IATC) se calcula restando el total de tablas de la base de datos, al total de accesos a las tablas que realiza el conjunto de Módulos (M) o Microservicios (MS), el resultado se divide entre el valor absoluto del total de módulos menos la multiplicación del número de tablas de la base de datos, multiplicada por el número de módulos del sistema. Por lo anterior la formula se representa de la siguiente manera.

El Índice de Acoplamiento por Tablas Compartidas (IATC) se calcula a partir de la diferencia entre el total de accesos a las tablas de la base de datos realizados por el conjunto de Módulos (M) o Microservicios (MS), y el número total de tablas existentes. Este resultado se divide entre el valor absoluto de la diferencia entre el número total de módulos y el producto del número de tablas por el número de módulos del sistema. La fórmula correspondiente se expresa de la siguiente manera:

$$IATC = \frac{\text{accTab} - NUT}{|NCU - (NUT * NCU)|}$$

Esta métrica se fundamenta en la teoría de medición propuesta por Zuse en [6] y se clasifica como una métrica de intervalo. Esto significa que no posee un cero absoluto, ya que no representa la ausencia total de la propiedad medida en este caso, el acoplamiento, sino que se construye a partir de un valor mínimo necesario y un valor máximo posible.

Las métricas de tipo intervalo permiten establecer un orden entre los objetos, es decir, determinar cuál presenta una mayor o menor cantidad de la propiedad analizada. Además, permiten realizar comparaciones relativas, por ejemplo: "la diferencia de acoplamiento entre la aplicación 1 y la aplicación 2 es de 0.33 unidades normalizadas".

Interpretación de los Valores del IATC

- Se establece el límite donde 0 es el valor menos acoplado y > 0 es el índice de acoplamiento.
- Un valor mayor a 0 de IATC indica un mayor nivel de acoplamiento indirecto en el sistema debido a tablas de base de datos compartidas.
- Un valor de $IATC = 0$ implica que no existen tablas compartidas en el sistema (cada tabla es accedida, como máximo, por un único componente). Este es el estado ideal en una arquitectura de microservicios pura donde cada servicio tiene total propiedad sobre sus tablas.

Sustento Teórico

$$IATC = \frac{\text{accTab} - NUT}{|NCU - (NUT * NCU)|}$$

Donde:

- IATC: Índice de acceso a tablas compartidas.
- accTab: Total de accesos únicos a las tablas de la base de datos.
- NUT: Número total de tablas.
- NCU: Número total de casos de uso.

Esta fórmula permite capturar no sólo la cantidad de accesos, sino también la redundancia en el uso de las tablas, lo que incrementa el acoplamiento. El resultado de la métrica tiende a 0 cuando

el acoplamiento es bajo (cada módulo accede a sus propias tablas), y aumenta a medida que el número de accesos compartidos crece.

En el estudio [6], los autores definen un marco matemático riguroso para evaluar métricas de software a través de la teoría formal de la medición, aplicando:

- Teoría axiomática de medición.
- Medidas como funciones que asignan números a propiedades de objetos del software.
- Condiciones necesarias: representación, validez, comparabilidad y significatividad semántica.

Zuse y Bollmann [6] adaptan axiomas de la teoría general de medición a métricas de software. Los más relevantes son:

- Completitud: Se puede decir que el sistema $A = B > C$
- Transitividad: El sistema $A \geq B, B \geq C \Rightarrow A \geq C$
- Asociatividad: $A = \text{contribución de Tabla1} + \text{Tabla2}$ (la aplicación de la métrica del sistema A se suma con la aplicación de la métrica del sistema B)
- Arquímedes: La diferencia entre A y C es 2; no hay valores infinitesimales ni discontinuos.
- Escala de razón: $A/C = \text{infinito}$ (acoplamiento total vs ningún acoplamiento), $B/C = \text{infinito}$
- Independencia: Agregar una tabla con $F_i = 1$ no cambia los valores de las otras tablas.

El Índice de Acoplamiento entre Tablas Compartidas (IATC) es una métrica que cuantifica el grado en que los módulos acceden a tablas comunes. Esta métrica permite identificar zonas de alto acoplamiento, potencialmente candidatas a refactorización o descomposición.

Nivel de Medición: Escala de Intervalo

Según la teoría de Stevens sobre los niveles de medición de variables, una métrica puede ser clasificada como:

- Nominal
- Ordinal
- De intervalo
- De razón

El IATC no parte de un cero absoluto, sino que su valor 0 no significa ausencia total de relación o acoplamiento, sino un punto de acoplamiento mínimo (ideal) en el contexto del sistema. Por tanto:

- El IATC se considera una métrica en escala de intervalo, ya que:
 - Permite ordenar los valores (más acoplado vs. menos acoplado).

- Las diferencias entre los valores son significativas (por ejemplo, un IATC de 0.4 implica mayor acoplamiento que 0.1, y la diferencia entre 0.3 y 0.1 es la misma que entre 0.5 y 0.3).
- El cero no implica una ausencia absoluta de acoplamiento, por lo tanto, no se trata de una escala de razón.

Propiedades de la Métrica

- Rango definido: El IATC está acotado en un intervalo definido, por ejemplo, de 0 a 1 (dependiendo de cómo se normalicen los valores).
- Interpretabilidad: Valores más cercanos a 0 representan menor acoplamiento entre módulos y tablas compartidas.
- Comparabilidad: Permite comparar entre diferentes módulos o casos de uso para identificar aquellos con mayor dependencia estructural en la base de datos.

Aplicación Práctica

Esta métrica puede ser utilizada para:

- Evaluar el acoplamiento de candidatos a microservicios durante una migración.
- Detectar cuellos de botella estructurales en el diseño de bases de datos monolíticas.
- Apoyar en la toma de decisiones arquitectónicas, minimizando la interdependencia entre módulos.

Conclusión

El IATC es una métrica de escala de intervalo, útil para medir el nivel de acoplamiento indirecto entre módulos mediante el acceso compartido a tablas de base de datos. Su correcta interpretación permite evaluar qué tan acoplado está un sistema en términos de su acceso a datos, apoyando esfuerzos de refactorización y migración hacia arquitecturas más desacopladas como microservicios.

Anexos C – Herramienta informática

A continuación, se presenta la interfaz de usuario de la herramienta informática que implementa el método de descomposición de la base de datos para operar en una arquitectura de microservicios.

TECNM · Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) | Departamento de Ciencias Computacionales

Herramienta de tesis doctoral

Inicio Resultados Acoplamiento

Identificador Automático de Microservicios

Herramienta de tesis · Ciencias Computacionales

Analiza un monolito Java mediante ANTLR y genera automáticamente los microservicios candidatos con modelo de datos, puertos y patrón Observador.

✓ Análisis estático ANTLR ✓ Generación Maven automática ✓ Patrón Observador ✓ Swagger UI incluido

Cargar monolito Java

Ingresar la ruta al directorio fuente del proyecto

Ruta del directorio fuente

C:\microservice-extractor\sample-monolith\src\main\java

Examinar...

Ejemplo: c:\microservice-extractor\sample-monolith\src\main\java

Analizar monolito

Monolito de ejemplo incluido

El módulo sample-monolith contiene un sistema de ventas (Customers, Orders, Products). Apunta la ruta a src/main/java

¿Cómo funciona?

- Parseo ANTLR**
Lee cada .java y construye el árbol sintáctico abstracto (AST)
- Visitor pattern**
Identifica @RestController, @Service, @Repository y entidades @Entity
- Detección de candidatos**
Cada método de Controller + 1 caso de uso = 1 microservicio candidato
- Patrón Observador**
Detecta entidades compartidas y genera observer / publisher automáticamente
- Generación Maven**
Crea proyectos Spring Boot listos para ejecutar con Swagger UI y H2

TECNM · Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) | Departamento de Ciencias Computacionales

Herramienta de tesis doctoral

Inicio Resultados Acoplamiento

Microservicios identificados

← Nuevo análisis

11
ARCHIVOS
PROCESADOS

2
CONTROLADORES
ENCONTRADOS

3
ENTIDADES
JPA

6
CANDIDATOS A
MICROSERVICIO

3
ENTIDADES
COMPARTIDAS

★ Patrón Observador activado

Se detectaron entidades compartidas entre microservicios. Los candidatos marcados con ★ recibirán el patrón Observador generado.

★ PointTransaction → [EnrollCustomerService, GetAccountByCustomerService, AddPointsService, RedeemRewardService, ListAvailableRewardsService, CreateRewardService]

★ LoyaltyAccount → [EnrollCustomerService, GetAccountByCustomerService, AddPointsService, RedeemRewardService, ListAvailableRewardsService, CreateRewardService]

★ Reward → [RedeemRewardService, ListAvailableRewardsService, CreateRewardService]

8 CANDIDATOS IDENTIFICADOS

1 EnrollCustomerService

LoyaltyController.enrollCustomer()

Puerto 8081 BD enrollcustomerdb Swagger UI

ENTIDADES JPA: LoyaltyAccount, PointTransaction

2 GetAccountByCustomerService

LoyaltyController.getAccountByCustomer()

Puerto 8082 BD getaccountbycustomerdb Swagger UI

ENTIDADES JPA: LoyaltyAccount, PointTransaction

3 AddPointsService

LoyaltyController.addPoints()

Puerto 8083 BD addpointsdbs Swagger UI

ENTIDADES JPA: LoyaltyAccount, PointTransaction

4 RedeemRewardService

RewardController.redeemReward()

Puerto 8084 BD redeemrewarddb Swagger UI

ENTIDADES JPA: Reward, LoyaltyAccount, PointTransaction

localhost:8084/swagger-ui/index.html#/

Swagger **v3/api-docs** **Explore**

OpenAPI definition v0 OAS 3.0

/v3/api-docs

Servers

<http://localhost:8084> - Generated server url

Observer Endpoints internos para sincronización entre microservicios

- POST** `/api/observer/product/updated` [Interno] Recibir notificación: Product actualizado en otro microservicio
- POST** `/api/observer/product/deleted` [Interno] Recibir notificación: Product eliminado en otro microservicio
- POST** `/api/observer/product/created` [Interno] Recibir notificación: Product creado en otro microservicio

Orders Gestión de órdenes de venta

Schemas

- PlaceOrderRequest >
- Customer >
- Order >

TECNM - Centro Nacional de Investigación y Desarrollo Tecnológico (CENIDET) | Departamento de Ciencias Computacionales

MicroserviceExtractor Identificador automático de microservicios

Herramienta de tesis doctoral

Inicio Resultados Acoplamiento

Métricas de Acoplamiento — IATC

Índice de Acoplamiento Tablas-Casos de Uso: mide cuántas tablas accede cada caso de uso en el monolito, comparado con el microservicio generado (que idealmente accede solo a su propia tabla).

$$IATC = (accTab - NUT) / |NCU - (NUT \times NCU)|$$

accTab = accesos totales a tablas NUT = tablas únicas del sistema NCU = número de casos de uso IATC = 0 → acoplamiento mínimo (ideal MS) IATC > 0 → mayor acoplamiento (monolito)

Ver resultados

Comparativa IATC — Monolito vs. Microservicios

Los tres monolitos de prueba del dominio de ventas

IATC antes (monolito) y después (microservicios) — TECNМ - CENIDET

Sistema	IATC Monolito	IATC Microservicios
M1 Sistema de Ventas	~0.5	~0.35
M2 Facturación y Devoluciones	~0.65	~0.35
M3 Fidelización de Clientes	~0.95	~0.65

Monolito 1 — Sistema de Ventas

C:\microservice-extractor\sample-monolito\src\main\java

30.0% reducción