UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Método DISC: Separando sistemas en microservicios"

TESIS

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:

ROBERTO PEDRAZA COELLO

TUTOR

Dr. Jorge Luis Ortega Arjona

Facultad de Ciencias

Ciudad Universitaria, CDMX a octubre de 2021

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto

En el año 2000, la compañía Netflix utiliza un sistema Web monolítico para renta de DVDs. Éste es modificado por múltiples personas diariamente. El sistema es desplegado una o dos veces a la semana con sus respectivos cambios. Si uno de los cambios ocasiona un problema en el sistema, es difícil y tardado diagnosticar la causa. Los componentes del sistema están profundamente interconectados, ocasionando que el equipo de desarrollo sufra por la poca agilidad con la que pueden trabajar. Por estos problemas y otros, Netflix decide adoptar el estilo arquitectónico de Microservicios para su negocio [9].

El estilo arquitectónico de microservicios ó MSA se basa en desarrollar una aplicación como un conjunto de pequeños servicios, donde cada uno de estos pequeños servicios es llamado un microservicio. MSA constituye un enfoque de la arquitectura de software que se basa en el concepto de modularización enfatizando las fronteras técnicas [15].

Cada microservicio es implementado y operado como un sistema independiente. Los microservicios ofrecen acceso a su funcionalidad y datos a través de una interfaz de red bien definida. Esta arquitectura mejora la agilidad del software porque cada microservicio se vuelve una unidad independiente de desarrollo, despliegue, operaciones, versiones y escalamiento [15].

MSA se ha vuelto la nueva tendencia en el desarrollo de software. Esto ha llevado a que compañías reconocidas mundialmente como Netflix, Amazon y Ebay hayan migrado a esta arquitectura. Sin embargo MSA no es una fórmula mágica y tiene varios desafíos en diferentes fases del desarrollo [27].

Una de las preocupaciones principales durante la etapa de diseño en el desarrollo de aplicaciones MSA es el dimensionamiento del servicio y la correcta separación de dominios. A menudo es difícil identificar las capacidades de negocio que deben asignarse a microservicio [20, 13]. En caso de no definir las fronteras de los microservicios de manera correcta, realizar una refactorización de éstas es difícil. No existe una herramienta de soporte para la refactorización de las fronteras y no se ha reportado una experiencia exitosa [4].

En el artículo de Ghofrani [13] se pregunta a profesionales que han trabajado con este estilo arquitectónico "¿Cómo seleccionas las fronteras de los microservicios?". En el artículo se observa que la mayoría de los profesionales con poca experiencia lo hacen manualmente, basándose en su experiencia y sus habilidades. Conforme los años de experiencia aumentan, se muestra que el uso de métodos sistemáticos para definir las fronteras de los microservicios tiende a tener más presencia. El método sistemático más mencionado es Diseño Impulsado por Dominio (*Domain Driven Design* o DDD).

DDD ayuda a observar el sistema desde la perspectiva de dominio de negocio, con el fin de separar el sistema en microservicios. Sin embargo, hay otras perspectivas para separación de servicios que DDD no toma en cuenta. Estas perspectivas deberían ser analizadas, principalmente si los requerimientos no funcionales así lo indican. Por ejemplo, si un sistema tiene una funcionalidad que requiere alto desempeño, sería importante analizar la arquitectura del sistema desde una perspectiva de calidad; no solo desde el dominio del negocio.

1.2. Problema

El problema que esta tesis busca resolver es apoyar a arquitectos de software a seleccionar qué funcionalidad y qué datos deben ser modelados juntos en un mismo

microservicio. Esto es visto como una preocupación según la literatura porque es complicado encontrar la granularidad correcta de un microservicio.

> "Con cualquier enfoque de modularización, encontrar los módulos correctos, con los tamaños correctos, la correcta asignación de responsabilidades e interfaces bien diseñadas, es un desafío. Esto es especialmente cierto para los microservicios y otros enfoques en lo que los límites mal diseñados pueden conducir a una mayor comunicación en la red."[1][15]

> "Durante el diseño de MSA, la falta de una guía de descomposición efectiva parece ser el aspecto más irritante para los profesionales." [2][27]

> "[. . .] al diseñar una aplicación basada en microservicios, uno de los problemas principales es determinar la granularidad "correcta" de los microservicios y el diseño de las políticas de seguridad."[3][20]

> "El dimensionamiento de servicios es significablemente reconocido como un problema, 16 de los estudios industriales seleccionados reconocen las dificultades para determinar qué tan "micro" debe ser un microservicio. Todos estos estudios coinciden en que a menudo es difícil identificar las capacidades de negocio/contexto acotado que debe asignarse a cada microservicio."[4][20]

> "Es difícil determinar exactamente donde deberían estar los límites de los componentes. El diseño evolutivo reconoce las dificultades de acertar

[1] With any approach to modularization, finding the right modules, with the right size, the right assignment of responsibilities, and well-designed interfaces, is a challenge. This is especially true for microservices and other approaches in which badly designed boundaries can lead to increased network communication.[15]

[2] During the design of MSA, the missing of an effective decomposition guide seems to be the most painful aspect for practitioners.[27]

[3] [. . .] while designing a microservice-based application, the primary pains are determining the right granularity of its microservices and the desing of its security policies.[20]

[4] Service dimensioning is also significantly recognized as a pain, with 16 of the selected industrial studies recognizing difficulties in determining how much "micro" a microservice should be. All such studies agree that it is often difficult to identify the business capability/bounded context that should be assigned to each microservice.[20]

los límites y, por lo tanto, la importancia de que sea fácil refactorizarlos. Pero cuando sus componentes son servicios con comunicaciones remotas, la refactorización es mucho más difícil que con las librerías en proceso."[5][12]

Existen otros métodos de descomoposición que pueden ser adoptados por la industría para identificar fronteras en la práctica de microservicios. Sin embargo, estos presentan características que se pueden mejorar. Algunas de estas características son:

- No presentan un proceso ordenado.
- No toman en cuenta los requerimientos no funcionales del sistema.
- Definen la granularidad de un microservicio como el tamaño funcional de éste. Sin embargo, utilizan métodos incorrectos para medir el tamaño funcional, aún cuando existen estándares internacionales para medir funcionalidad de software.
- La notación que se utiliza para ingresar los datos al método es complicada y, por lo tanto, es propensa a errores.

La hipótesis de esta tesis es:

Se propone generar un método que ayude a separar un sistema en microservicios. Este método utiliza los requerimientos funcionales a un nivel de granularidad de proceso funcional y los requerimientos no funcionales para apoyar al arquitecto de software al momento de tomar decisiones sobre qué procesos funcionales pertenecen a un mismo microservicio, y sobre qué grupos de datos son persistentes en cada microservicio.

1.3. Aproximación

El presente trabajo introduce un método para separar un sistema en microservicios, mediante el análisis de los requerimientos funcionales, a un nivel de granularidad

[5] It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries.[12]

de proceso funcional, junto con los requerimientos no funcionales. A éste método se le ha asignado el nombre "Método DISC".

El método DISC se apoya en los criterios de acoplamiento: Proximidad semántica, Dueño compartido, Volatilidad estructural, Latencia, Criticalidad de consistencia, Disponibilidad crítica, Semejanza de almacenamiento, Tráfico similar en la red, Contextualidad de seguridad y Criticalidad de la seguridad. Estos criterios de acoplamiento son mencionados en la literatura como desencadenantes de decisiones en arquitecturas basadas en microservicios. [14]

Añadido a esto, el método propone utilizar conceptos del estándar internacional de medición de tamaño funcional de software COSMIC para medir la granularidad de los microservicios, así como el acoplamiento entre microservicios.

1.4. Contribuciones

Las contribuciones del presente trabajo son:

- **Método con un proceso ordenado:** El método DISC ofrece un proceso ordenado, cosa que no hacen muchos de los otros métodos que buscan separar un sistema en microservicios.
- **Método que toma en cuenta los requerimientos no funcionales del sistema:** El método DISC toma en cuenta los requerimientos no funcionales del sistema al momento de separar un sistema en microservicios.
- **Concepto de granularidad de microservicios objetivo:** Este concepto es mencionado múltiples veces en la literatura. Sin embargo, son pocas las definiciones que se pueden encontrar. Las definiciones que se encuentran son subjetivas. Esta tesis propone utilizar conceptos de COSMIC para definir la granularidad de un microservicio.
- **Métricas objetivas para microservicios:** Esta tessis presenta una forma de medir granularidad de microservicios y una forma de medir acoplamiento entre

microservicios. Ambas métricas se basan en conceptos del método de medición de COSMIC.

- **Método expandible:** La naturaleza del método DISC es tal que, si un arquitecto de software considera necesario agregarle más tareas a un subproceso o un nuevo subproceso, es posible hacerlo.

1.5. Estructura de la tesis

El presente trabajo está separado en capítulos, los cuales se describen brevemente a continuación:

Capítulo 2 **Antecedentes**

En este capítulo se presentan los conceptos necesarios para entender el contexto de la tesis. Primero, se conceptualizan conceptos generales de MSA. Despues, se explica cuáles son los tipos de microservicios y las características de estos. Posteriormente, se explica el método de DDD. En seguida, se presentan los criterios de acoplamiento en los que se basa el método presentado en esta tesis. Posteriormente, se dan a conocer patrones arquitectónicos que se consideran importantes en la práctica de microservicios. Después de esto, se presentan malas prácticas que se observan en la industria al desarrollar aplicaciones MSA. A continuación, se explican algunos conceptos básicos del método de medición de software COSMIC. Finalmente, se presenta la notación BPMN y algunos de sus elementos.

Capítulo 3 **Trabajo relacionado**

En este capítulo se describe el trabajo relacionado a los métodos existentes para identificar microservicios. Primero se presenta el artículo *"Requirements Reconciliation for Scalable and Secure Microservice (De)composition"*, en el cual se muestra una metodología que permite separar un sistema en microservicios tomando en cuanto requerimientos no funcionales. Posteriormente, se presenta el artículo *"Identifying Micro-services Using Functional Decomposition"*, que define un enfoque sistemático para identificar microservicios en una etapa temprana de diseño. Finalmente, se presenta el artículo *"Service Cutter: A Systematic Approach to Service Decomposition"*, el cual presenta un método, basado en algoritmos de agrupamiento y que toma en cuenta requerimientos funcionales y no funcionales del sistema, para proponer microservicios.

Capítulo 4 **Método DISC: Separando sistemas en microservicios**

En este capítulo se describe el método DISC. Se da una pequeña introducción del problema que se busca resolver. Se explican las 5 etapas del método (Analizar perspectiva de dominio, Analizar perspectiva de Infraestructura, Analizar perspectiva de Seguridad, Analizar perspectiva de Calidad y Resolver contradicciones) de manera resumida. Posteriormente, se explica cada una de las etapas del método a detalle. Al final del capítulo, se presenta una propuesta que utiliza conceptos del método de medición COSMIC, para medir el acoplamiento entre microservicios y la granularidad de los microservicios.

Capítulo 5 **Casos de Estudio**

En este capítulo se desarrollan 2 casos de estudio utilizando el método DISC. Se realizan las tareas de los procesos y subprocesos para obtener una salida para cada caso de estudio. Las salidas del método DISC se comparan con las salidas del método Service Cutter [14]. Para los casos de estudio, el método DISC y el método Service Cutter utilizan las mismas fuentes de información como entrada. Por esto último, se considera válido realizar la comparación entre los dos métodos.

Capítulo 6 **Conclusiones**

En este capítulo se presentan las conclusiones de la tesis. Primero se ofrece un pequeño resumen de la tesis. En seguida, se enuncian las contribuciones de la tesis. Posteriormente, se presentan las ventajas y desventajas del método DISC. Finalmente, se mencionan un par de temas que pueden ser interesantes como trabajo futuro de esta tesis.

Capítulo 2

Antecedentes

En este capítulo se presentan los conceptos necesarios para entender el contexto de la tesis. Primero, se explican conceptos generales del estilo arquitectónico de microservicios. Despues, se mencionan cuáles son los tipos de microservicios. Posteriormente, se explica el método Domain-Driven Design. En seguida, se presentan algunos criterios de acoplamiento mencionados en la literatura como desencadenantes de decisiones en arquitecturas basadas en microservicios. Posteriormente, se dan a conocer patrones arquitectónicos que se consideran importantes en la práctica de microservicios. Después de esto, se presentan malas prácticas que se observan en la industria al desarrollar aplicaciones MSA. A continuación, se explican algunos conceptos básicos del método de medición de software COSMIC. Finalmente, se presenta la notación BPMN y algunos de sus elementos.

2.1. Microservicios

En el blog escrito por Fowler y Lewis se describe a la **Arquitectura de Microservicios (MSA)** [1] [12] de la siguiente manera:

[1] The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

> "El estilo arquitectónico de microservicios es un enfoque para desarrollar una aplicación como un conjunto de pequeños servicios, cada uno corriendo en su propio proceso y comunicándose con mecanismos ligeros, usualmente una API de recursos HTTP. Estos servicios son construidos basándose en las capacidades del negocio y se pueden desplegar independientemente por mecanismos de despliegue completamente automatizados. Existe un mínimo de gestión centralizada de estos servicios, los cuales pueden estar escritos en diferentes lenguajes de programación y usar diferentes tecnologías de almacenamiento de datos."

Para comprender mejor el estilo de microservicios, es útil compararlo con el estilo monolítico. Una aplicación monolítica usualmente consiste de tres partes: Una aplicación del lado del cliente, una base de datos y una aplicación del lado del servidor. Esta aplicación del lado del servidor es un monolito, es decir, una única unidad ejecutable. Todo cambio a la aplicación del lado del servidor implica construir y desplegar toda una nueva versión [12].

En cambio, con MSA, cada microservicio es implementado y operado como un pequeño sistema independiente, ofreciendo acceso a su funcionalidad interna y datos por medio de una interfaz de red. Esto mejora la agilidad del software porque cada microservicio se convierte en una unidad independiente de desarrollo, despliegue, operación, versionamiento y escalamiento [15].

En la Figura 2-1, se observa como una aplicación monolítica debe ser replicada totalmente en múltiples servidores para escalarla; a diferencia de los microservicios donde se puede escalar cada servicio en servidores diferentes según se necesite.

La idea general de MSA es generar un aplicación como un conjunto de servicios interconectados entre ellos. El término de **Servicio** puede ser definido tanto en un nivel lógico como en un nivel físico.

1. "Un servicio es la autoridad técnica para una capacidad de negocio específica." [7]

2. Un servicio es parecido a un componente, ya que es utilizado por aplicaciones

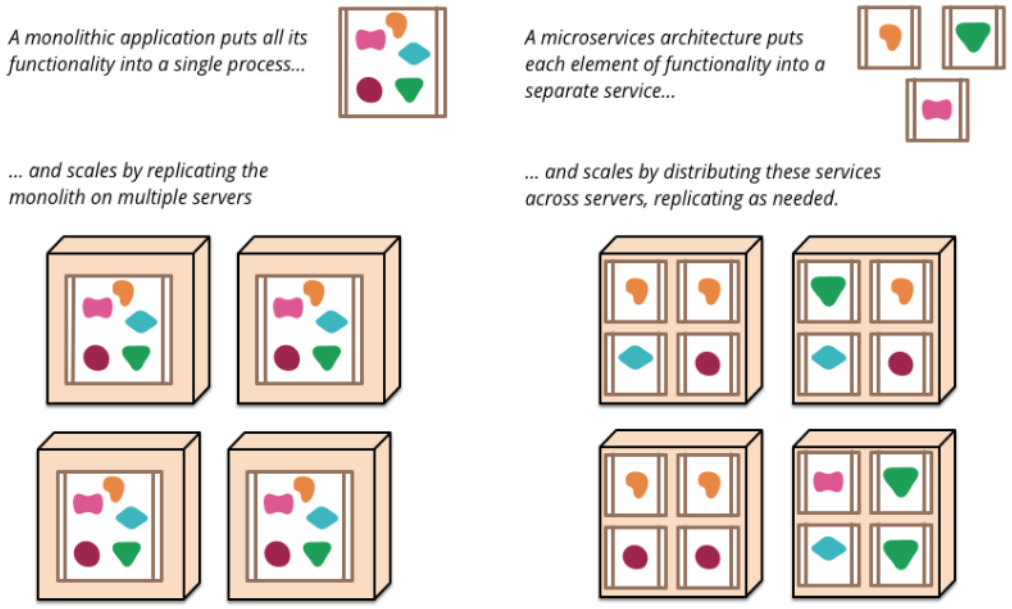

Figura 2-1: Monolitos y Microservicios [12].

foráneas. La diferencia es que el servicio es accesado por medio de una interfaz remota; se espera que el componente se utilice localmente, por ejemplo: un archivo jar o dll[10].

Un servicio necesita recursos para proveer capacidades a un usuario. El concepto de **nanoentidad** generaliza los 3 tipos de recursos que un servicio puede requerir [14]:

- **Datos.** Un servicio puede ser dueño de un subconjunto de datos del sistema.
- **Operaciones.** Un servicio encapsula reglas de negocio y lógica de procesamiento. Las operaciones usualmente se basan en los datos que posee el servicio.
- **Artefactos.** Es una vista de datos o operaciones transformada a un formato específico. Por ejemplo, un reporte de las ventas mensuales, el cual fue creado utilizando operaciones y datos.

Los microservicios se pueden desarrollar con diferente tamaño en cuanto a su funcionalidad. Una definición de **Granularidad** de microservicio utilizada en la literatura [19] menciona que la granularidad es lo mismo que el tamaño de la funcionalidad. Esta misma definición menciona que el tamaño de la funcionalidad puede ser la

complejidad del código o el número de casos de uso implementados en un microservicio. Sin embargo, (como se explica en la sección 4.7.1 de esta tesis) el concepto de granularidad es subjetivo si se plantea de esta manera.

2.1.1. Tipos de Microservicios

Algunos individuos y organizaciones han intentado clasificar los diferentes tipos de microservicios. Algunos autores clasifican los microservicios en dos tipos (Stateless y Stateful) [9]. Otros autores mencionan que hay tres tipos [2]:

- **Stateless.** Este tipo de microservicios realizan procesos que no dependen de otros microservicio para ejecutar sus tareas porque todo el funcionamiento y datos que estos necesitan lo tienen ellos mismos. Estos no escriben datos ni leen datos de ningun almacenamiento persistente. Al contar con un bajo acoplamiento con otros microservicios, son los más faciles de reemplazar y de escalar [2].

  El código de ejemplo que se presenta a continuación muestra el tipo de funciones que un microservicio debe tener para ser considerado stateless.

  Código ejemplo:

```
//Método que convierte grados Celsius a grados
    Fahrenheit
public float conversionCelsiusToFahrenheit(float
    celsius){
     return (celsius * 9/5) + 32;
}
```

- **Data-centric.** Se suele referir a esta clasificación de microservicios como un sistema de registros. Estos realizan escrituras y/o lecturas en almacenamiento persistente. Al igual que los microservicios stateless sus funciones no dependen de otros microservicios. No se realizan commits en dos fases, es decir, cuando

se guarda un registro en la base de datos no se manda información a otro microservicio para que guarde algun otro registro o realice alguna tarea [2].

El código de ejemplo que se presenta a continuación muestra el tipo de funciones que este tipo de microservicios contiene.

Código ejemplo:

```
//Método que actualiza una orden de compra
public boolean updateOrder(Order order){
    //Se actualiza la orden en el almacenamiento
       local, como una base de datos.
    boolean response = localStorage.update(order)
       ;
    return response;
}
```

- **Aggregator.** Este tipo de microservicos depende de otros microservicios para ejecturar sus tareas. Los microservicios aggregator, al estar fisicamente separados del resto de los microservicios, dependen de la red para funcionar correctamente. Estos microservicios pueden o no contar con almacenamiento persistente [2].

  Código ejemplo:

```
/**
* Este método actualiza una orden en su base de datos
   .
* Posteriormente manda la información de la orden a
   otros
* microservicios para que hagan sus procesos
   correspondientes.
*/
public boolean updateOrder(Order order){
    //Se actualiza la base de datos
    boolean response = orderStorage.update(order);
```

```
        //Lista de los recursos que la función necesita.
        List<Resource> resourceList = ["customer", "sales
            "];
        //Se obtienen la lista de microservicios que
            corresponden a los recursos.
        ServiceList serviceList = gatherResources(
            resourceList);
        if(response)
            //serviceList.orderUpdated se encarga de
                hacer llamadas a los servicios web de los
                otros microservicios.
            response = serviceList.orderUpdated(order);
        return response;
    }
```

2.1.2. Domain-Driven Design

Domain-Driven Design (DDD) es un método de descomposición que ayuda a los arquitectos de software a separar un sistema en microservicios. Es de los métodos más utilizados en la industria y es, hasta donde se sabe, al que más se hace referencia en la literatura [3, 13, 16]. Existen otros métodos de descomposición, como descomposición horizontal y descomposición vertical, pero la explicación se centrará en DDD [27].

En el libro de Evans [8] se hace referencia a lo que el llama "sistemas grandes". No se da una definición sobre qué es un sistema grande, pero se da a entender que es un sistema que tiene mucha funcionalidad. Evans menciona que en la mayoría de los sistemas grandes se pueden observar diferentes modelos. Un **modelo** "es una intepretación de la realidad que abstrae los aspectos relevantes para resolver un problema e ignora los detalles extraños." El modelo general de un sistema grande está compuesto por varios modelos más pequeños que están entremezclados entre ellos [8].

El método de DDD se basa en separar el modelo general del sistema grande en modelos más específicos que representen un contexto de negocio. Un modelo exitoso debe ser logicamente consistente y no debe tener definiciones contradictorias [8].

Los sistemas empresariales algunas veces tienen subsistemas o aplicaciones tan distintas que es complicado englobarlas en un mismo dominio. Para resolver este problema se define un **contexto acotado** dentro del cual un cierto modelo aplica. Se pueden hablar diferentes "lenguajes" dependiendo del contexto acotado al cual se esté haciendo referencia [8].

Por ejemplo, un sistema de una librería tiene su dominio general que es la venta de libros. Puede haber un subdominio de administración de catálogo donde un libro puede ser representado como una portada, un título, un autor, etc. Puede existir otro subdominio de envíos donde un libro es una cierta dimensión, un cierto peso, el costo de la tarifa internacional, entre otros datos.

Se puede observar que, dentro de un mismo sistema, al salir del contexto acotado de envíos lo que se define como "libro" es muy diferente. Esta separación entre lo que está dentro de un contexto acotado y el resto del universo se define como **frontera**. Cuando se aplican fronteras siguiendo el libro de Evans [8] se esperan obtener contextos acotados.

Los conceptos de contexto acotado y **modelo de dominio** van de la mano al ser este último el modelo que aplica dentro de un contexto acotado. Es un diagrama que representa el conocimiento que tiene un experto de un cierto dominio. Se presenta el conocimiento selectivamente y de una manera rigurosamente organizada [8]. Por ejemplo, en una organización se pueden tener diferentes contexto (ventas, recursos humanos, desarrollo, etc). Por lo que se puede separar conceptualmente la organización en diferentes contextos (contextos acotados), y se pueden generar diagramas para cada uno de los contextos (diagrama de dominio). Estos diagramas deben representar el conocimiento

Los contextos acotados representan dominios de negocio autónomos. Por esto son un punto de inicio apropiado para las líneas divisoras de los microservicios. Se dice que utilizando DDD, con el enfoque de contextos acotados, las posibilidades de tener componentes altamente acoplados es baja [16].

2.1.3. Criterios de acoplamiento

Un criterio de acoplamiento representa una fuerza impulsora para la descomposición de servicios. Estos criterios capturan requerimientos arquitectónicamente significantes del porqué dos nanoentidades deben o no pertencer al mismo servicio [14].

Los defensores de microservicios sugieren utilizar DDD para obtener las fronteras de los servicios. Sin embargo, existen otras inquietudes de los participantes que se deben tomar en cuenta al momento de descomponer los servicios, particularmente requerimientos arquitectónicamente significantes, incluyendo atributos de calidad del software [14].

En el artículo de Gysel [14] se presenta un catálogo de 16 criterios de acoplamiento separados en 4 perspectivas y 4 categorías como se observa en la figura 2-2. Las 4 categorias se pueden explicar de la siguiente manera:

1. **Cohesividad:** Criterios que describen propiedades comunes de nanoentidades mutualmente relacionadas, que justificaría porque estas entidades deben pertenecer al mismo servicio. Por ejemplo, todas las nanoentidades involucradas en la realización de un caso de uso deberían pertenecer al mismo servicio [14].

2. **Compatibilidad:** Criterio que indica características divergentes de las nanoentidades. Un servicio no debería contener nanoentidades con características incompatibles. Por ejemplo, hay datos que deben ser fuertemente consistentes y otros que pueden soportar inconsistencias. Las nanoentiedades que manejan estos no deberían estar en un mismo servicio [14].

3. **Restricción:** Criterios que especifican requerimientos de alto impacto que obligan a que un grupo de nanoentidades (a)deba pertenecer a un mismo servicio o (b)deban pertenecer a diferentes servicios. Por ejemplo, si el cambio de valor de un atributo involucra el cambio de valor de otro, esos dos atributos deberían ser responsabilidad del mismo servicio [14].

4. **Comunicación:** Criterios pertenecientes exclusivamente al costo técnico de llamadas remotas. Por ejemplo, las nanoentidades pequeñas y poco frecuentadas

son más adecuadas para ser compartidas entre servicios [14].

Además de estas cuatro categorías, cada uno de los criterios pertenecen a una perspectiva, las cuales pueden ser: perspectiva de dominio, perspectiva de calidad. perspectiva física y perspectiva de seguridad. A continuación se presentan los 16 criterios de acoplamiento de Gysel:

Categorías / Perspectivas	Cohesividad	Compatibilidad	Restricción	Comunicación
Dominio	CC-1 Identidad y Comunalidad del Ciclo de vida CC-2 Proximidad Semántica CC-3 Dueño compartido	CC-4 Volatilidad Estructural		
Calidad	CC-5 Latencia	CC-6 Criticalidad de Consistencia CC-7 Disponibilidad Crítica CC-8 Volatilidad del Contenido	CC-9 Restricción de Consistencia	CC-10 Mutabilidad
Fisica		CC-11 Semejanza de Almacenamiento	CC-12 Restricciones predefinidas del servicio	CC-13 Tráfico Similar en la Red
Seguridad	CC-14 Contextualidad de Seguridad	CC-15 Criticalidad de la Seguridad	CC-16 Restricción de Seguridad	

Figura 2-2: Catálogo de los 16 criterios de acoplamiento, clasificados por categorias y perspectivas. Adaptado de [14].

- **CC-1 Identidad y Comunalidad del Ciclo de vida.** Este presenta la idea de juntar en un mismo servicio las nanoentidades que pertenezcan a la misma identidad. Por pertenecer a la misma identidad, comparten el mismo ciclo de vida. Por ejemplo, acoplar en un mismo servicio las clases que tengan una relación de herencia o composición [14].

  Este criterio es de **tipo cohesividad** ya que este tipo de relaciones entre nanoentidades explícitamente hace que compartan propiedades. También pertenece a la **perspectiva de dominio** ya que, al existir este tipo de relación en el sistema, tiene sentido que el mismo tipo de relación exista en la organización que va a utilizar el software [14].

- **CC-2 Proximidad Semántica.** El criterio expone la idea de que las nanoentidades que cuenten con proximidad semántica pertenezcan al mismo micro-

servicio. Dos nanoentidades son semánticamente próximas si comparten una conexión semántica dada por el dominio de negocio. Por ejemplo, cuando dos nanoentidades deben ser accedidas en el mismo caso de uso [14].

Este criterio de acomplamiento es de **tipo cohesividad** ya que, al necesitar varias nanoentidades en un mismo caso de uso, se puede suponer una relación entre éstas. También pertenece a la **perspectiva de dominio** ya que la conexión semántica está dada por el mismo dominio de negocio [14].

- **CC-3 Dueño compartido.** Este criterio introduce el razonamiento de que las nanoentidades que sean responsabilidad de un mismo dueño estén en un mismo servicio. Un dueño puede ser, por ejemplo, una persona, un rol o un departamento [14].

  Este criterio es de **tipo cohesividad** ya que el fin es mantener junto todo lo referente a un dueño. La propiedad que comparten las nanoentidades es el dueño. De igual manera pertenece a la **perspectiva de dominio** ya que la separación que se hace tiene que ver con la estructura de trabajo de la organización [14].

- **CC-4 Volatilidad estructural.** El criterio presenta que se debe tomar en cuenta qué tan seguido se piden cambios al código de una nanoentidad. Buscando separar las nanoentidades que cambien frecuentemente de otras que raramente cambian [14].

  Este criterio es de **tipo compatibilidad** porque busca separar estas nanoentidades que no son compatibles de manera que el realizar muchos cambios en una nanoentidad no tenga efectos colaterales en una nanoentidad que no cambia. El criterio pertenece a la **perspectiva de dominio** porque la frecuencia de cambio de una nanoentidad depende de cómo funciona la organización [14].

- **CC-5 Latencia.** Este criterio busca que los grupos de nanoentidades con requerimientos de alto desempeño que realizan un proceso específico se encuentren en el mismo microservicio. Esto con el fin de evitar llamadas remotas y, con esto, evitar latencias [14].

Este criterio es de **tipo cohesividad** ya que busca mantener juntas las nanoentidades que necesitan la propiedad de tener un alto desempeño. Pertenece a la **perspectiva de calidad** porque intenta mejorar el desempeño del sistema [14].

- **CC-6 Criticalidad de consistencia.** Este criterio de acoplamiento se basa en que algunos datos, como registros financieros, pierden su valor cuando tienen inconsistencias, mientras que otros datos son más tolerantes a inconsistencias [14].

  Este criterio busca que las nanoentidades incompatibles desde el punto de criticalidad de consistencia no se encuentren en el mismo servicio. Esto lleva a decir que el criterio es de **tipo compatibilidad**. El criterio se basa en la consistencia, que es un factor de calidad, por esto pertenece a la **perspectiva de calidad** [14].

- **CC-7 Disponibilidad Crítica.** El criterio menciona que las nanoentidades incompatibles en cuestión de disponibilidad no deben estar en el mismo servicio. Tener una disponibilidad alta tiene un costo en desarrollo y en producción, por lo que las nanoentidades con diferentes requerimientos de disponibilidad no deben estar en un mismo microservicio [14].

  El criterio busca separar nanoentidades incompatibles en cuanto a disponibilidad. Esto lo hace del **tipo compatibilidad**. De igual manera, se enfoca en cuestiones de disponibilidad que es un factor de calidad del software. Por esto pertenece a la **perspectiva de calidad** [14].

- **CC-8 Volatilidad del Contenido.** Este criterio busca separar las nanoentidades que son incompatibles según la volatilidad de su contenido. La volatilidad se refiere a qué tan seguido se actualiza el contenido de las nanoentidades, por ejemplo, qué tan seguido se actualiza la base de datos [14].

  Se recomienda que las nanoentidades que son altamente volátiles y las nanoentidades que son más estables no pertenezcan al mismo servicio, por esto el criterio es de **tipo compatibilidad**. Se dice que el criterio pertenece a la **perspectiva**

de calidad ya que, mientras más volatiles sean los datos de la nanoentidad, más problemas puede haber respecto a la calidad de los datos [14].

- **CC-9 Restricción de Consistencia.** Este criterio busca mantener juntas las nanoentidades que comparten un estado y deben tener la propiedad de permanecer consistentes unas con otras. Esta propiedad se observa como un conjunto de objetos que son tratados como una unidad con el propósito de cambiar datos. Si modificar el valor de un atributo involucra modificar el valor de otro, estos dos atributos deben ser responsabilidad del mismo servicio [14].

  Este criterio representa una restricción de requerimientos que hace que un grupo de nanoentidades deba estar en el mismo servicio. Por esto, el criterio es de **tipo restricción**. Este criterio busca la consistencia de información en un grupo de nanoentidades, por esto pertenece a la **perspectiva de calidad** [14].

- **CC-10 Mutabilidad.** Este criterio busca separar las nanoentidades que manejen información inmutable. La información inmutable es más simple de manejar en un sistema distribuido. Se menciona que la descomposición de servicios debe ser realizada de una manera que favorezca el compartir las nanoentidades inmutables. Las nanoentidades inmutables son buenas candidatas a ser un lenguaje mutuo entre dos servicios [14].

  Este lenguaje común mejora la comunicación entre los dos microservicios. Por esto es de **tipo comunicación**. Este lenguaje inmutable y mutuo entre dos microservicios genera un bajo acoplamiento entre estos. Esto lleva a decir que el criterio pertenece a la **perspectiva de calidad** [14].

- **CC-11 Semejanza de Almacenamiento.** Este criterio busca separar las nanoentidades basándose en el almacenamiento necesario para persistir las instancias de las nanoentidades. Es decir, si una nanoentidad persiste archivos de 1 GB y otra nanoentiedad persiste archivos de 1 MB, éstas no deberían estar juntas [14].

  Una nanoentidad que requiere un alto nivel de almacenamiento necesita una

infraestructura diferente a otra que no lo requiere. Por esto el criterio pertenece a la **perspectiva de infraestructura**. Se busca separar las nanoentidades que no son compatibles, por esto mismo, el criterio es de **tipo compatibilidad** [14].

- **CC-12 Restricciones predefinidas del servicio.** Se puede dar el caso de que algunas nanoentidades forzosamente deban ser modeladas juntas. Por ejemplo, para lograr una optimización tecnológica o porque hay un sistema heredado que se va a utilizar. Esto es una restricción del proyecto, por esto mismo el criterio es de **tipo restricción**. Esta restricción afecta como se despliega el sistema, por esto pertenece a la **perspectiva de infraestructura** [14].

- **CC-13 Tráfico Similar en la Red.** Hay nanoentidades que se comparten entre servicios. Este criterio presenta la idea de que las nanoentidades pequeñas y menos frecuentadas son más adecuadas para ser compartidas entre servicios. Al ser poco frecuentadas, el tráfico de la red es poco afectado, por esto el criterio es de **tipo comunicación**. El poner estas nanoentidades en un microservicio compartido modifica cómo se comporta la red, por esto este criterio pertenece a la **perspectiva de infraestructura** [14].

- **CC-14 Contextualidad de Seguridad.** El criterio busca que si un rol de seguridad es capaz de ver o procesar un grupo de nanoentiedad, éstas se encuentren en un mismo servicio. Mezclar contextos de seguridad en un servicio complica implementar la autenticación y autorización [14].

  Estas nanoentidades tendrían en común la propiedad de ser accesadas por un mismo rol. Por esto, este criterio es de **tipo cohesividad**. El punto de este criterio es no complicar la autenticación y autorización para evitar problemas de seguridad, lo que lleva a decir que el criterio pertenece a la **perspectiva de seguridad** [14].

- **CC-15 Criticalidad de la Seguridad.** En algunas nanoentidades es crítico no perder datos o no tener violaciones de seguridad. Estos niveles de seguridad tienen un costo de desarrollo y procesamiento, por lo que nanoentidades con di-

ferentes niveles de criticalidad no deben pertenecer al mismo servicio; el criterio es de **tipo compatibilidad** y pertenece a la **perspectiva de seguridad** [14].

- **CC-16 Restricción de Seguridad.** Este criterio explica que puede haber un grupo de nanoentidades semánticamente relacionadas que no deben recidir en el mismo servicio por requerimientos de seguirdad. Estas restricciones pueden ser dadas por un tercero. Por ejemplo, una autoridad de certificación. Al ser presentada la restricción de mantener nanoentiedades separadas, este criterio es de **tipo restricción**. Al centrarse en factores de seguridad, el criterio pertenece a la **perspectiva de seguridad** [14].

2.2. Patrones arquitectónicos

Los desarrolladores de software tienden a tomar decisiones arquitectónicas basándose en experiencias anteriores o en los patrones más populares de Internet, incluso si las decisiones no son las más apropiadas [23, 24]. Por esto se presenta una introducción a algunos patrones arquitectónicos útiles en MSA.

- **Patrón API Gateway.** Este patrón permite generar un punto de entrada hacia la aplicación desde el exterior y enrutar las peticiones hacia los microservicios. Si, durante la ejecución de un proceso funcional, un cliente requiere información o funcionalidad de varios microservicios, este patrón permite que el cliente realice una sola petición. El API Gateway se encarga de realizar la composición de las respuestas y responder al usuario. Es parecido al patrón *Fachada* ya que encapsula la arquitectura interna de la aplicación y provee una API para los clientes [18]. Ver figura 2-3.

- **Patrón de descubrimiento del lado del cliente.** La función de este patrón es poder generar una comunicación dinámica entre microservicios y sus clientes. Este dinamismo es necesario para que los clientes puedan encontrar las múltiples instancias de un mismo microservicio [24].

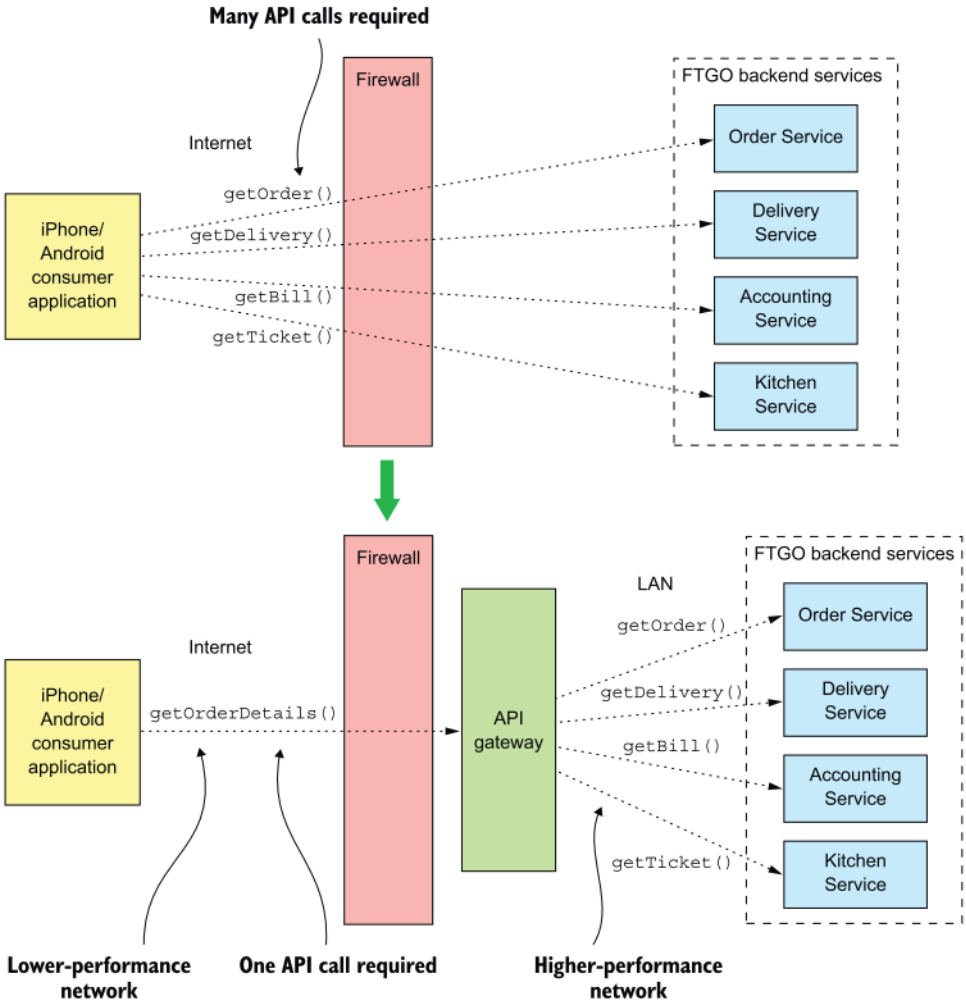

Figura 2-3: Un API Gateway permite que un cliente web, en este caso un dispositivo móvil, obtenga datos realizando una sola solicitud [18].

Este patrón busca que al generar una nueva instancia de un microservicio, ésta se registre en un *registro de servicios*. El cliente hace una consulta a este *registro de servicios* para obtener las direcciones de las instancias del microservicio. El cliente hace el trabajo de balanceador de carga seleccionando, de alguna manera, a qué instancia hacer la petición. Con esto se puede generar dinámicamente una comunicación directa entre el cliente y el microservicio [24]. Ver figura 2-4.

- **Patrón de descubrimiento del lado del servidor.** Este patrón tiene la misma función que el patrón de descubrimiento del lado del cliente. La principal diferencia entre estos dos es que el patrón de descubrimiento del lado del

servidor realiza la consulta al *registro de servicios* con un balanceador de carga como mediador entre estos dos. Este balanceador de carga regresa al cliente la dirección de la instancia a la cual hacer la petición [24]. Ver figura 2-4.

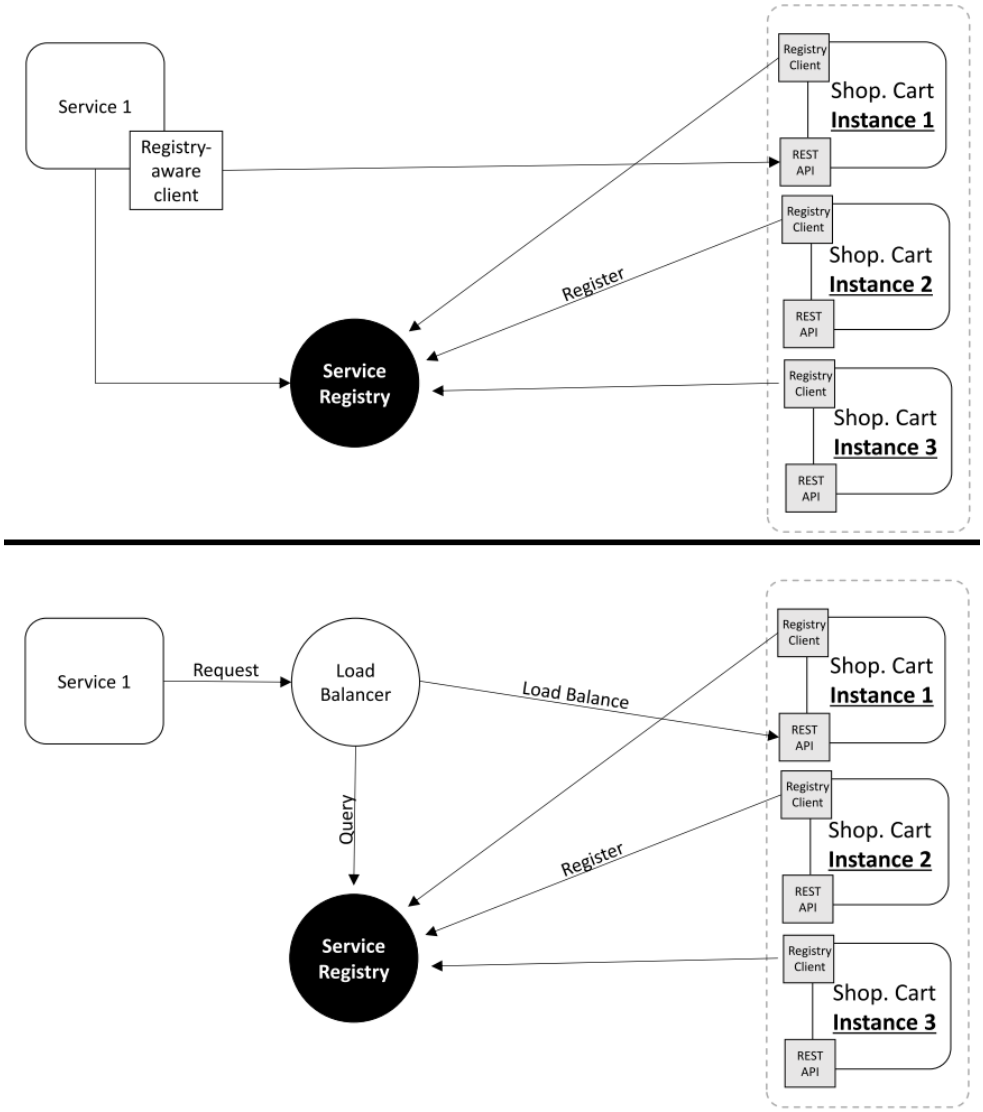

Figura 2-4: Patrón de descubrimiento del lado del cliente (arriba) y patrón de descubrimiento del lado del servidor (abajo) [24].

- **Patrón de múltiples servicios por nodo.** Este patrón introduce la idea de tener múltiples servicios corriendo en un mismo nodo. En la academia se observa que un enfoque muy utilizado es tener cada servicio en su propio contenedor y múltiples contenedores en un mismo nodo. La idea contraria (tener un solo

servicio por nodo) afecta el rendimiento y escalabilidad del sistema [24].

- **Patrón de base de datos compartida.** La idea de este patrón es que varios microservicios accedan a una sola base de datos. Esto es útil en cierto casos. Por ejemplo, cuando una funcionalidad deba asegurar una invariante entre varios microservicios. Esto ayuda a reforzar la consistencia de datos, pero genera acoplamiento entre los microservicios. La documentación del patrón menciona que un microservicio puede acceder a los datos que son propiedad de otro microservicio solo mediante transacciones ACID [17].

- **Patrón de base de datos por servicio.** La idea de este patrón es que cada microservicio tenga su propia base de datos, donde sus datos solo pueden ser accesados desde el exterior por medio de la API del microservicio. Esto ayuda a asegurar un bajo acoplamiento entre los microservicios, pero genera complejidad al momento de querer mantener la consistencia en los datos [17].

2.3. Malas prácticas

La meta de esta tesis es presentar un método que apoye en la toma de decisiones al diseñar sistemas MSA. Por esto se considera importante conocer las malas prácticas o problemas que se presentan habitualmente en la industria.

Las malas prácticas que se exponen se obtienen de dos artículos. En el primero, [22], se realizan entrevistas a desarrolladores con experiencia en microservicios. De las entrevistas se obtuvo un conjunto de malas prácticas en MSA y se explican maneras de superar estas malas prácticas. El segundo artículo, [3], introduce malas prácticas y soluciones propuestas para las mismas. En este último artículo, las malas prácticas y soluciones se han obtenido de 58 fuentes de la academia y literatura gris (documentos que no han pasado por una revisión por pares para su publicación).

En esta sección solo se presentan malas prácticas que afectan, ya sea por el contexto del problema o por la solución propuesta, la arquitectura del sistema. El resto de malas prácticas, aunque es importante conocerlas, quedan fuera del enfoque de

esta tesis.

- **No tener un API Gateway.** Esta mala práctica se presenta cuando los microservicios se comunican directamente uno con otro. También se puede considerar cuando el cliente-web se comunica directamente con los microservicios. Esto causa que se incremente la complejidad del sistema, afectando la mantenibilidad del mismo. La solución que se presenta es aplicar el patrón *API Gateway* para reducir la complejidad de comunicación entre microservicios [22].

- **Dependencia cíclica.** Este es un problema que se describe como una cadena cíclica de llamadas entre microservicios. Por ejemplo, A llama a B, B llama a C y C llama a A. Este tipo de ciclos ocasiona que los microservicios sean difíciles de mantener o reutilizar. Se menciona que la manera de solucionar este problema es aplicar el patrón *API Gateway* [22].

- **Intimidad inapropiada en servicio.** Este problema se puede observar cuando un microservicio se conecta seguido a otro servicio para obtener datos privados, en vez de manejar sus propios datos. Por ejemplo, un servicio que requiera conectarse directamente a la base de datos de otro servicio. Esto aumenta el acoplamiento entre los dos microservicios. El problema puede estar relacionado a un error de cómo se modelaron los datos. Se debe considerar unir los microservicios en uno solo para deshacerse de este tipo de relaciones [22].

- **Microservicio codicioso.** Esta mala práctica se da cuando el equipo de desarrollo crea nuevos microservicios para cada función del software, incluso cuando no es necesario. Este tipo de microservicios se caracterizan por tener funcionalidades muy limitadas. Esta práctica puede llevar a tener un número muy grande de microservicios. Un sistema con muchos microservicios se vuelve más difícil de mantener. Se recomienda considerar cuidadosamente si un nuevo microservicio es necesario o no [22]. Se menciona también en [3], que un microservicio con una granularidad demasiado fina o demasiado gruesa puede producir problemas.

- **Bibliotecas compartidas.** Esta mala práctica se presenta cuando dos microservicios comparten una misma biblioteca. Esto puede causar problemas si la biblioteca se actualiza. En ese caso, los equipos se deben coordinar para modificar la biblioteca que comparten. Las soluciones posibles son: 1) Aumentar la dependencia entre equipos y aceptar la redundancia, o 2) crear un nuevo servicio con las funciones de la biblioteca [22].

- **Persistencia compartida.** Esta mala práctica se introduce cuando diferentes microservicios acceden a una misma base de datos causando un alto acoplamiento entre los microservicios. Al acceder a los mismos datos, la independencia de los servicios y de los equipos de desarrollo se reduce. En este caso se presentan 3 soluciones posibles: 1) Utilizar bases de datos independientes para cada servicio, 2) utilizar una misma base de datos con un conjunto de tablas privadas que solo puedan ser accesadas por un servicio, o 3) usar un esquema de base de datos privado para cada servicio [22].

- **Demasiados estándares.** MSA permite adoptar diferentes lenguajes y tecnologías para diferentes microservicios. Sin embargo, utilizar demasiadas tecnologías es considerado una mala práctica para las empresas. La solución recomendada es considerar cuidadosamente si es necesario adoptar una nueva tecnología para un microservicio [22].

- **Cortes erróneos.** Esta mala práctica se observa cuando un sistema es separado en microservicios tomando como base las capas técnicas (presentación, negocio y datos) en vez de separar por capacidades de negocio. La solución presentada es realizar un análisis claro de las funciones de negocio y los recursos necesarios para realizar esas funciones [22]. Este mismo problema se presenta en [3] donde se menciona que, si los equipos son separados por capas técnicas, un cambio simple requiere más tiempo y esfuerzo para que se apruebe entre los equipos.

- **Contendor codicioso de servicio.** Los contenedores permiten desplegar aplicaciones de manera autónoma. No utilizarlos añade complejidad a la admi-

nistración y orquestración de la aplicación. Sin embargo, configurar los contenedores toma tiempo. Se podría pensar en usar un solo contenedor para toda la aplicación. Esto es considerado una mala práctica ya que el contenedor se vuelve pesado y reduce la independencia entre los servicios. La solución es utilizar un contenedor por servicio. Esto permite una reducción en el tiempo para desplegar, y mejora la capacidad de escalar el sistema [3].

2.4. COSMIC

El método COSMIC (ISO/IEC 19761:2017) provee un método estandarizado para medir el tamaño funcional del software [5].

En la sección 2.1 se menciona el concepto de granularidad de microservicio. Se dice que la granularidad de un microservicio es el tamaño de la funcionalidad del mismo. De igual manera, se menciona que el tamaño de la funcionalidad puede ser equivalente a la complejidad del código, el número de casos de uso o el número de nanoentidades implementadas en un microservicio.

El problema que se encuentra con esto es que no hay una manera estandarizada de detallar los casos de uso, la complejidad o las nanoentidades. Esto puede llevar a que cierta funcionalidad en un microservicio sea un solo caso de uso para una persona, mientras para otra persona pueden ser 2 o más casos de uso. Por lo que, para un mismo microservicio, diferentes personas podrían tener diferentes perspectivas en cuanto la granularidad del mismo.

Por esto, se propone utilizar el concepto de proceso funcional que presenta el método COSMIC para definir la granularidad. De tal manera que el número de procesos funcionales que pertencen a un microservicio represente la granularidad del microservicio. Un proceso funcional es un conjunto de movimientos de datos que representa una parte elemental de los requerimientos funcionales del usuario. El proceso funcional es único entre los requerimientos funcionales del usuario y puede definirse independientemente de cualquier otro proceso funcional [5].

Cada proceso funcional debe estar detallado a un nivel de granularidad donde:

- sus usuarios funcionales (tipos) sean humanos individuales, dispositivos de ingeniería, piezas de software (no grupos de usuarios funcionales) [5].

- ocurren eventos únicos (tipos) a los que la pieza de software debe responder (no a un nivel de granularidad en el que se definen grupos de eventos) [5].

Un ejemplo de "Grupos de usuarios funcionales" es el departamento de una empresa cuyos miembros, dependiendo de su puesto, manejan diferentes tipos de procesos funcionales. Un "Grupo de eventos" puede estar declarado como un requerimiento de alto nivel. Por ejemplo, en un software de contabilidad, un requerimiento descrito como "transacciones de ventas". Existen diferentes tipos de transacciones de ventas, y existen eventos únicos a los que el software debe responder con transacciones de ventas diferentes [5].

Habiendo identificado los procesos funcionales, lo siguiente que se debe hacer es identificar los movimientos de datos. Durante un movimiento de datos se mueve un grupo de datos, cuyos atributos de datos describen un objeto de interés. Por lo que, para entender el concepto de movimiento de datos, se deben definir otros 3 conceptos [5].

- Un objeto de interés es cualquier cosa en el mundo del usuario funcional que se identifica en los requerimientos funcionales del usuario del software que se va a medir. Puede ser una cosa física, así como un objeto conceptual o parte de un objeto conceptual [5].

- Un grupo de datos es un conjunto único, no vacío y no ordenado de atributos de datos, donde cada atributo de dato incluido describe un aspecto complementario del mismo objeto de interés. Cada grupo de datos debe ser único y distinguible a través de su colección de atributos de datos [5].

- Un atributo de datos es la forma más pequeña de información dentro de un grupo de datos identificado [5].

Teniendo los grupos de datos identificados, lo siguiente que se debe hacer es identificar los movimientos de datos que se realizan durante un proceso funcional. COSMIC

define 4 movimientos de datos: Entrada, Salida, Lectura y Escritura. Un movimiento de datos de entrada se presenta cuando se mueve un solo grupo de datos desde el usuario funcional hacia el proceso funcional. Un movimiento de datos de salida se presenta cuando un grupo de datos se mueve desde el proceso funcional hacia el usuario funcional. Un movimiento de datos de lectura mueve un grupo de datos desde un almacenamiento persistente hacia el proceso funcional. Un movimiento de datos de escritura se presenta cuando un grupo de datos se mueve desde el proceso funcional hacia el almacenamiento persistente [5].

Los conceptos de COSMIC utilizados por el método DISC son: proceso funcional, grupo de datos y movimiento de datos.

2.5. Modelo y Notación de Procesos de Negocio

La notación la notación *Business Process Model and Notation* (BPMN) fue desarollada por Object Management Group para describir la lógica de los pasos de un proceso de negocio. La notación incluye varios elementos para modelar. Sin embargo, en esta tesis no se explican todos los elementos, solo algunos de estos [26].

- **Tarea.** Representa una actividad atómica que está incluida dentro de un proceso [26]. Ver figura 2-5 (a)
- **Subproceso Colapsado.** Los detalles de un subproceso no son visibles en el diagrama [26]. Ver figura 2-5 (b)
- **Subproceso Expandido.** La frontera del subproceso se expande y los detalles (del subproceso) son visibles dentro de su frontera [26]. Ver figura 2-5 (c)
- **Evento de Inicio.** Indica dónde empieza un proceso (o subproceso) particular [26]. Ver figura 2-5 (d)
- **Evento de Fin.** Indica dónde termina un proceso (o subproceso) [26]. Ver figura 2-5 (e)

- **Flujo de secuencia.** Usado para mostrar el orden en que las actividades se realizan en el proceso [26]. Ver figura 2-5 (f)
- **Puerta de enlace paralela.** Se utiliza para crear flujos paralelos o para sincronizar (combinar) flujos paralelos [26]. Ver figura 2-5 (g)
- **Objeto de datos.** Provee información que necesitan las actividades para realizarse, o indica la información que la actividad producen [26]. Ver figura 2-5 (h)
- **Asociación.** Usado para enlazar información o artefactos con un elemento gráfico de BPMN. La punta de flecha indica el flujo [26]. Ver figura 2-5 (i)
- **Anotación de Texto.** Mecanismo para proveer información adicional al lector del diagrama BPMN [26]. Ver figura 2-5 (j)

2.6. Resumen

En este capítulo se presenta una breve introducción a la arquitectura basada en microservicios. Se identifican los 3 tipos de microservicios que existen y las características de estos mismos. Se mencionan algunos métodos de descomposición utilizados frecuentemente en la industria, dando énfasis al método de descomposición *Domain-Driven Design* (DDD) por ser el método que más se utiliza en la industria. También se presentan varios criterios de acoplamiento que se deben tomar en cuenta al diseñar un sistema MSA.

Posteriormente se explican unos cuantos patrones arquitectónicos. Aunque los patrones no son especificos de MSA, estos resuelven problemas que comúnmente se encuentran al diseñar software basados en microservicios. Despues, se presentan malas prácticas que se han observado en la industría y soluciones para estas mismas. En seguida, se explican conceptos básicos del método de medición de software COSMIC. Finalmente, se explica la notación BPMN y ciertos elementos utilizados para realizar diagramas.

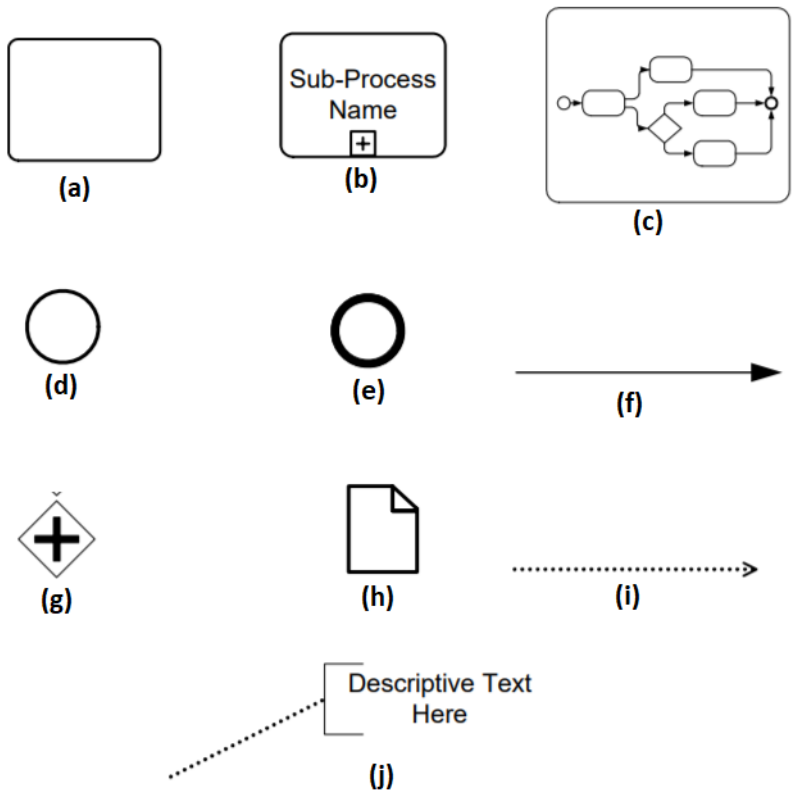

Figura 2-5: Elementos pertenecientes a la notación BPMN [26].

Capítulo 3

Trabajo Relacionado

Este capítulo presenta los trabajos relacionados con el tema de esta tesis. Se analizan los artículos: 1) Conciliación de requerimientos para las descomposición de microservicios escalables y seguros (*Requirements Reconciliation for Scalable and Secure Microservice (De)composition*), 2) Identificación de microservicios utilizando descomposición funcional (*Identifying Microservices Using Functional Decomposition*) y 3) Cortador de servicios: Un enfoque sistemático para la descomposición de servicios (*Service Cutter: A Systematic Approach to Service Decomposition*).

3.1. Conciliación de requerimientos para las descomposición de microservicios escalables y seguros

Las decisiones que se toman al momento de separar un sistema en microservicio normalmente son tomadas por desarrolladores o arquitectos con algunas intuiciones de como hacerlo. En el artículo "*Requirements Reconciliation for Scalable and Secure Microservice (De)composition*" [1] se argumenta que este enfoque solo captura el interés de un conjunto reducido de los interesados, es decir, no toma en cuenta todos los requisitos del sistema. Por esto se propone una metodología conceptual que toma en cuenta requerimientos de seguridad y escalabilidad para la descomposición de

microservicios.

Es importante construir la arquitectura del sistema de una manera que mapee los requerimientos funcionales a microservicios, tomando en cuenta la escalabilidad y seguridad. Con esto en mente, se presenta la metodología que se muestra en la figura 3-1.

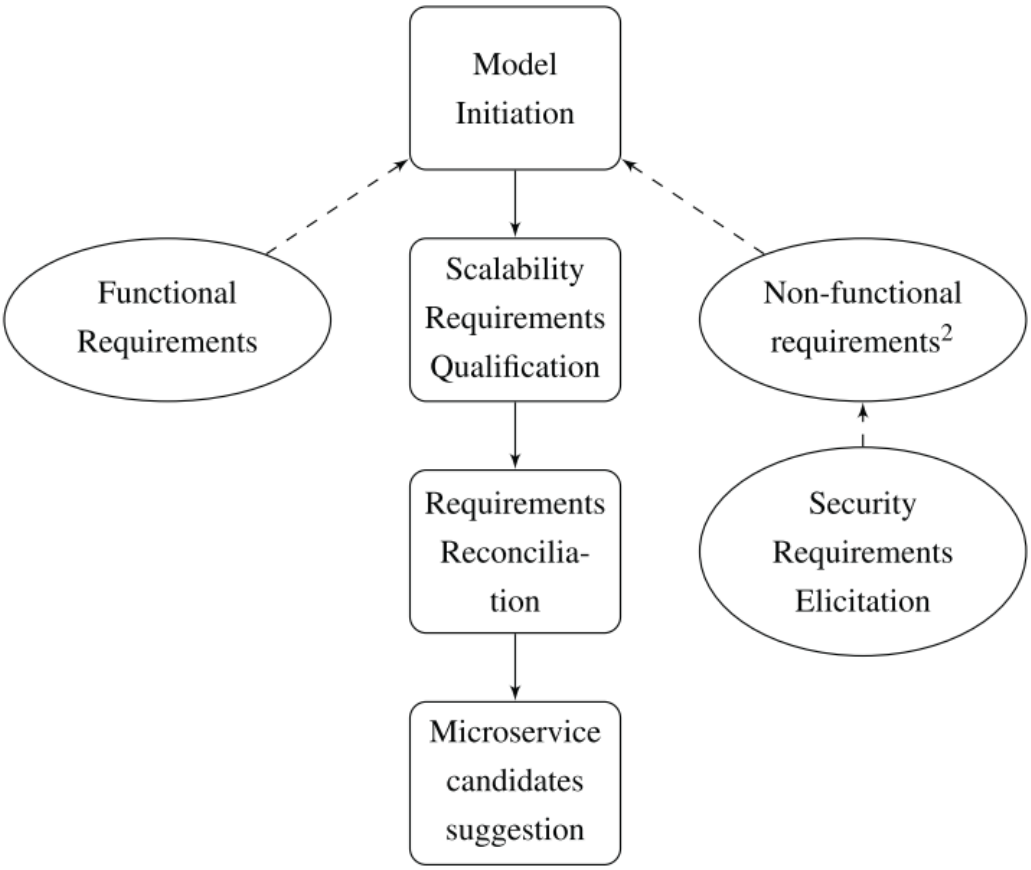

Figura 3-1: Metodología de descomposición de Microservicios [1].

Se asume que un sistema que se quiere diseñar se expresa por medio de un conjunto de requerimientos funcionales y no funcionales. Pueden existir dependencias entre un requerimiento funcional y otro. Por cada dependencia se debe obtener algo que el artículo define como *peso de dependencia*. El peso de dependencia se refiere a qué tan fuerte es una dependencia entre dos requerimientos funcionales. Por ejemplo, el peso de dependencia entre facturación y un carrito de compras es alto, porque cada llamada a facturación necesita una llamada a carrito de compras. En cambio si un

requerimiento funcional realiza una llamada a otro requerimiento una vez al día el peso de dependencia es bajo.

Una fase de esta metodología es la obtención de requerimientos de seguridad *(Security Requirements Elicitation)*. El artículo no presenta ni recomienda un proceso para obtener estos requerimientos. El mecanismo que se utilice debe generar una política de seguridad para el sistema. La metodología mapea estas políticas a los requerimientos funcionales correspondientes. Por ejemplo, se puede decidir agregar la política *P1*, que incluye encriptación y autorización, al requerimiento funcional de facturación, ya que este último accede a la información de tarjeta de crédito del usuario.

Habiendo terminado la fase de obtención de requerimientos se tiene un conjunto de requerimientos funcionales. A cada uno de estos se le han asignado un conjunto de políticas de seguridad específicas. En la fase de iniciación del modelo *(Model initiation)* se evalúa, para cada requerimiento funcional, el impacto de cumplir con las póliticas utilizando uno de estos niveles: alto, medio o bajo.

En la fase de calificación de requisitos de escalabilidad *(Scalability Requirements Qualification)* se determina el nivel de escalabilidad requerido para cada requerimiento funcional. Se utiliza una medida cualitativa para medir la escalabilidad deseada donde el ingeniero de requerimientos debe preguntarse *¿Cuántos usuarios simultáneos se espera que tenga esta funcionalidad?*. La respuesta debe ser dada en uno de estos niveles: alto, medio o bajo.

En la fase de conciliación de requerimientos *(Requirements Reconciliation)* se razona sobre el equilibrio de escalabilidad y seguridad como factores para componer un servicio. Se sigue el siguiente flujo secuencial.

1. Si el nivel de escalabilidad del requerimiento funcional *(FR)* es medio o alto, entonces iniciar un microservicio *MS* con este *FR*. Ver figura 3-2 línea 6.

2. Iterar sobre el conjunto de requerimientos funcionales relacionados (dependencias). Ver figura 3-2 líneas 8-14.

3. Si no se ha hecho todavía, evaluar el impacto de cumplir con las políticas de seguridad para cada dependencia.

4. Obtener el máximo entre el peso de la dependencia, el nivel de la política de seguridad del requerimiento y el nivel de la política de seguridad de la dependencia. Ver figura 3-2 línea 11.

5. Si el máximo obtenido en el paso 4 es mediano o alto, entonces agregue la dependencia a *MS*. Ver figura 3-2 línea 11-13.

```
1:  ML ← InitMicroserviceList
2:  FR ← FunctionalRequirement
3:  V ← DependantRequirement(FR)
4:  S ← ScalabilityRequirement(FR)
5:  P ← SecurityPolicy(FR.Assets, FR.Threats)
6:  if Scalability is a requirement of FR then
7:     MS ← FR
8:     for all v ∈ V do
9:        v_u ← dependency weight of the relation FR − v
10:       v_p ← security policy of the dependant requirement
11:       if Max(v_u, FR.P, v_p) = High then
12:          MS ← merge(FR, v)
13:       end if
14:    end for
15:    ML ← add(MS)
16: end if
```

Figura 3-2: Procedimiento de Conciliación [1].

Los autores reconocen que la seguridad y la escalabilidad no son los únicos factores que deben ser considerados en la descomposición de microservicios. Otros factores, como el costo y la mantenibilidad, no fueron estudiados en el artículo.

3.2. Identificación de microservicios utilizando descomposición funcional

Como se ha venido mencionando, un desafío que se presenta cuando se utiliza MSA es encontrar la correcta partición del sistema en microservicios. Usualmente

esto se hace de manera intuitiva, basándose en la experiencia de los diseñadores. El artículo "*Identifying Microservices Using Functional Decomposition*" [25] describe un enfoque sistemático para identificar microservicios en una etapa temprana de diseño. Este enfoque se basa en la especificación de los requerimientos funcionales del sistema y en la descomposición funcional de éstos.

Se identifican las relaciones entre las operaciones que se requieren del sistema y las variables de estado que esas operaciones escriben o leen. Con esto se busca poder visualizar las relaciones entre las operaciones del sistema y las variables de estado. Esta visualización permite encontrar particiones del espacio de estados con relaciones densas. Las particiones se vuelven candidatas a microservicios buscando que las operaciones de cada microservicio accedan, en la medida posible, solo a las variables de ese microservicio.

Para utilizar este enfoque se necesita crear un modelo del sistema. El modelo consiste de un conjunto finito de operaciones del sistema y el espacio de estados del sistema. Las operaciones del sistema son las operaciones públicas (métodos) del sistema. El espacio de estado es el conjunto de variables del sistema que contienen información que las operaciones del sistema leen y escriben.

En el artículo se obtienen las operaciones del sistema y las variables de estado basándose en casos de uso. Se identifican todos los verbos en los casos de uso como operaciones. Los sustantivos encontrados en la descripción de los casos de uso sirven como una aproximación a las variables de estado del sistema. Esta información es registrada en una tabla de operación/relación, donde se registra que variables son leídas o escritas por una operación.

La tabla de operación/relación se puede convertir a un grafo. Esto permite identificar grupos de relación densa que están débilmente conectados a otros grupos de relación densa. Cada uno de estos grupos es un buen candidato para convertirse en microservicio porque la cantidad de información que comparte con el resto del sistema es pequeña (bajo acoplamiento) y porque las relaciones internas son densas (alta cohesión).

Se construye un grafo no dirigido bipartito G cuyos vértices representan las varia-

bles de estado del sistema y sus operaciones. Las aristas conectan una operación *op* con una variable de estado *v* si y solo si *op* lee o actualiza *v*. Se le asigna un peso a cada arista. El artículo propone un peso de 1 para lectura y un peso de 2 para escritura. Esto con el fin de agrupar, en la medida posible, las operaciones que modifiquen una variable de estado. Es preferible una interfaz de lectura sobre una de escritura entre grupos de relación. La figura 3-3 muestra el resultado de realizar estas acciones para el caso de estudio presente en el artículo.

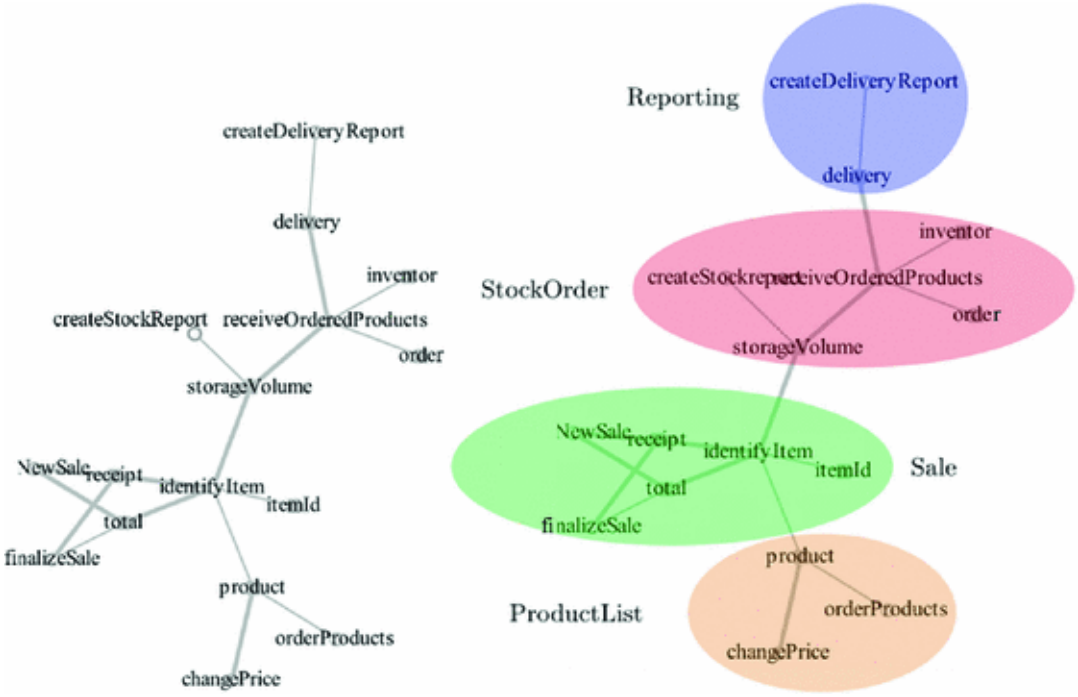

Figura 3-3: Un grafo de dependencia operación/relación. La parte izquierda muestra el diagrama antes de identificar los microservicios. La parte derecha presenta los microservicios (las formas de colores). Las aristas delgadas representan lectura y las gruesas escritura. [25].

Para evaluar este enfoque se han realizado implementaciones de microservicios por tres equipos independientes, descomponiendo el mismo sistema que se utiliza como caso de estudio en el artículo. Los resultados muestran que con este enfoque se realiza la descomposición más rapido que manualmente y que los resultados entre el diseño manual y este enfoque son comparables.

3.3. Cortador de servicios: Un enfoque sistemático para la descomposición de servicios

Es particularmente importante separar un sistema distribuido en unidades con poco acoplamiento y alta cohesión. La información que se tiene sobre como separar un sistema en servicios discretos y autónomos es vaga. Este artículo propone una manera estructurada y repetible sobre cómo realizar la descomposición de servicios. La descomposición se basa en los 16 criterios de acoplamiento mencionados en la sección 2.1.3 de esta tesis. Estos criterios de acoplamiento (ver figura 2-2) se obtuvieron de la literatura y de la experiencia en la industria [14].

Los criterios de acoplamiento son la base de "Service Cutter", el método propuesto por Gysel para la descomposición de servicios. Este método no busca automatizar completamete la toma de decisiones, más bien busca apoyarla. El usuario que utiliza este método, conociendo los requerimientos funcionales y no funcionales del sistema, debe generar un par de archivos llamados "artefactos de especificación de sistema" en formato JSON. Estos archivos contienen información sobre los casos de uso del sistema, las nanoentidades, los criterios de acoplamiento que aplican al sistema, etc. Estos archivos se procesan de forma semi-automática con el fin de sugerir una descomposición de servicios que promueva el bajo acoplamiento y alta cohesión de servicios [14].

El método recibe como entrada los "artefactos de especificación de sistema". Esta especificación del sistema permite generar un grafo pesado no dirigido. Los nodos representan nanoentidades (concepto mencionado en la sección 2.1 de esta tesis) y las aristas pesadas indican que tan acopladas son 2 nanoentidades. El método utiliza algoritmos de agrupamiento para encontrar cortes de servicios. La implementación del método incluye 4 algoritmos de agrupamiento [14]:

- **Leung.** Algoritmo de agrupamiento no determinista. Por lo que, para una misma entrada, se pueden obtener diferentes salidas.
- **Girvan-Newman.** Algoritmo de agrupamiento en el cual debe recibir como

entrada el número de grupos (en este caso microservicios) que tiene la salida.

- **Chinese Whisperers.** Algoritmo de agrupamiento aleatorio. Por lo que, para una misma entrada, se pueden obtener diferentes salidas.

- **Markov (MCL).** Algoritmo de agrupamiento. No se deben seleccionar el número de servicios, y es determinista. Por lo que una misma entrada siempre tiene una misma salida.

Para calcular el peso de una artista entre 2 nanoentidades se calcula el "puntaje de prioridad". Este puntaje se calcula de los puntajes de los criterios de acoplamiento (datos presentes en los "artefactos de especificación de sistema") multiplicado por las prioridades (estos datos se ingresan directamente en la implementación web). El ejemplo de cómo se calcula el peso de una arista se observa en la figura 3-4 [14].

Coupling criterion	Score	Priority	Result
CC-1: Semantic Proximity	4	1	$4 * 1 = 4$
CC-7: Availability Criticality	2.5	5	$2.5 * 5 = 12.5$
CC-9: Consistency Constraint	8	3	$8 * 3 = 24$
Total weight			$4 + 12.5 + 24 = 40.5$

Figura 3-4: Ejemplo del cálculo del peso de una arista [14].

Para validar este método se utilizaron 2 sistemas como casos de estudio (los mismos sistemas que se utilizan en los casos de estudio de esta tesis). Para que la validación sea valida Gysel define los cortes esperados, como línea base, para ambos sistemas. Los cortes que el método ofrece como salida se dividen en 3 categorías: corte excelente (mejor que el corte de la línea base), corte esperado (igual al corte de la línea base) y corte irrazonable (peor que el corte de la línea base; no se encuentra razones de porque ese corte beneficia). La salida del método se clasifica en 4 niveles: Excelente (ni un corte irrazonable y al menos uno excelente), bueno (ni un corte irrazonable), aceptable (a lo más un corte irrazonable) y malo (2 o más cortes irrazonables) [14].

Para el caso de estudio de "Trading System" se obtuvieron buenas salida con el algoritmo de Girvan-Newman y de Leung. Para el caso de estudio de "Cargo Tracking

System” se obtuvo una mala salida con el algoritmo de Girvan-Newman y una salida aceptable con el algoritmo de ELP. Un ejemplo de salida del caso de estudio “Trading System” se puede observar en la figura 3-5 [14].

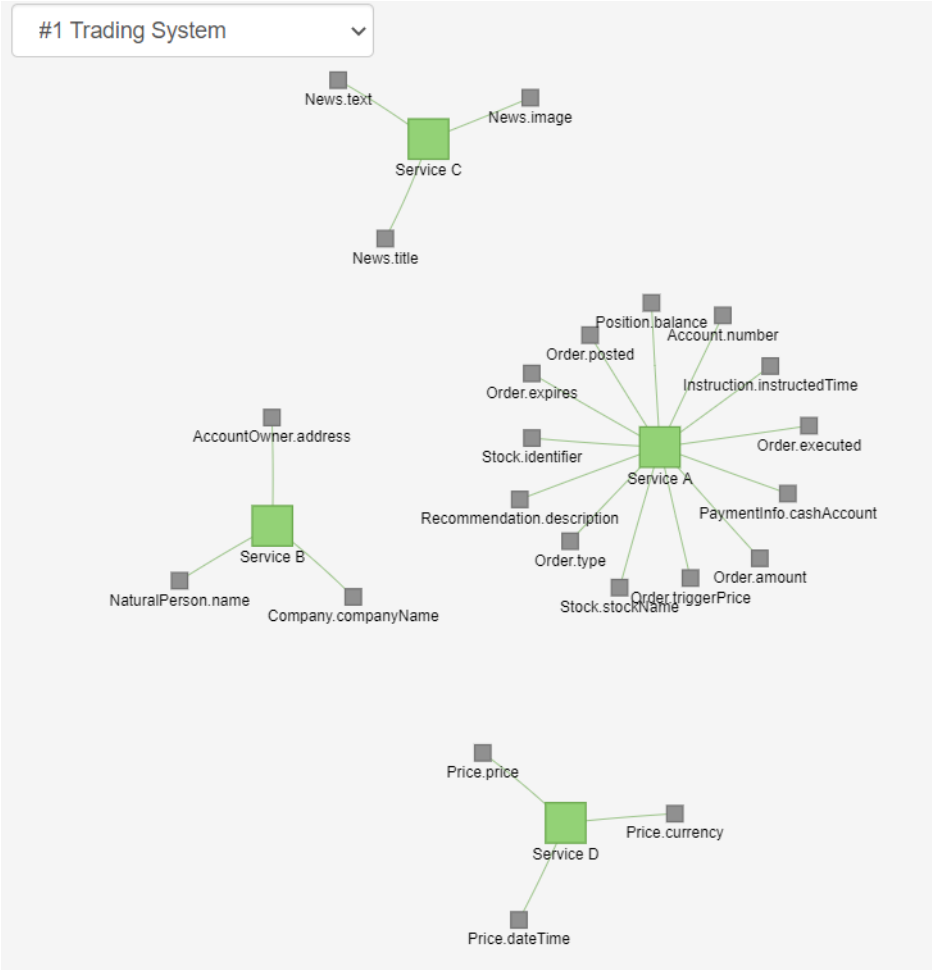

Figura 3-5: Salida del método “Service Cutter” para el caso de estudio “Trading System” [14].

3.4. Resumen

En este capítulo se presenta el resumen de tres trabajos relacionados con la identificación de microservicios para un sistema. Los tres trabajos buscan identificar qué funcionalidades van a pertenecer a un mismo servicio y qué funcionalidades van a pertenecer a diferentes servicios.

En el primer trabajo se realiza esto tomando en cuenta requerimientos no funcionales. Especificamente los requerimientos de escalabilidad y seguridad. El artículo presenta una metodología que permite realizar esta actividad de una manera sistemática.

El segundo artículo tambien presenta un enfoque sistemático para separar un sistema en microservicios. Este enfoque se basa en la descomposición funcional de los requerimientos funcionales. La idea que presenta es visualizar, en forma de grafo, las relaciones entre las operaciones y las variables. Con esto se busca encontrar, en el grafo, particiones con relaciones densas. Estas particiones son las que se identifican como candidatas a microservicios.

El último artículo presenta un método, basado en algoritmos de agrupamiento, que toma en cuenta requerimientos funcionales y no funcionales del sistema para proponer microservicios. También presenta un catálogo de criterios de acoplamiento que apoya la toma de decisiones arquitectónicas.

Capítulo 4

Método DISC: Separando sistemas en microservicios

En este capítulo se describe el método DISC, que es el método propuesto en esta tesis. Este método apoya al arquitecto de software en la tarea de separar un sistema en microservicios. Se utilizan los requerimientos funcionales a un nivel de granularidad de proceso funcional y los requerimientos no funcionales.

El capítulo está organizado de la siguiente manera, primeramente se explican las 5 etapas del método (Analizar la perspectiva de dominio, Analizar la perspectiva de Infraestructura, Analizar la perspectiva de Seguridad, Analizar la perspectiva de Calidad y Resolver contradicciones) de manera resumida. Posteriormente se explica cada una de las etapas del método a detalle. Al final se presentan propuestas sobre cómo medir granularidad y acoplamiento en microservicios.

La finalidad del método que se propone es ser un apoyo en la toma de decisiones del arquitecto de software al separar un sistema en microservicios. La separación del sistema se realiza tomando en cuenta los requerimientos funcionales y no funcionales del sistema.

Como se menciona en la sección 2.1, la granularidad de un microservicio se refiere al tamaño de la funcionalidad. Puede ser la complejidad del código, el número de casos de uso o el número de nanoentidades implementadas en un microservicio. El problema que se encuentra con esto es que no hay una manera estandarizada de

detallar los casos de uso, la complejidad o las nanoentidades. Por esto, el método que aquí se propone utiliza el concepto de proceso funcional presentado por COSMIC. De tal forma que la granularidad de un microservicio se define mediante el número de procesos funcionales de un microservicio.

4.1. Diseño del método

Este método está estructurado en 5 etapas: Analizar la perspectiva de Dominio, Analizar perspectiva de Infraestructura, Analizar perspectiva de Seguridad, Analizar perspectiva de Calidad, y Resolver Contradicciones (Se nombró método DISC por los 4 análisis de perspectivas).

La primera etapa (Analizar perspectiva de Dominio) se enfoca en juntar en un mismo microservicio procesos funcionales que tengan características en común en cuanto al dominio del sistema. Las siguientes 3 etapas (Analizar perspectiva de Infraestructura, Analizar perspectiva de Seguridad y Analizar perspectiva de Calidad) reciben como entrada la salida de la etapa pasada. Estas 3 etapas se pueden realizar en paralelo. En la última etapa (Resolver Contradicciones), se reciben las salidas de las 3 etapas pasadas y, de existir contradicciones entre las salidas, éstas son resueltas. Las 5 etapas se pueden observar en la figura 4-1 y se pueden resumir de la siguiente manera:

1. **Analizar la perspectiva de Dominio.** Este subproceso comienza creando un microservicio para cada proceso funcional. Haciendo esto se descompone todo el sistema en microservicios de granularidad 1 (un solo proceso funcional por microservicio). Estos microservicios se comienzan a fusionar si los procesos funcionales escriben el mismo grupo de datos, pertenecen al mismo caso de uso o pertenecen al mismo dueño.

2. **Analizar la perspectiva de Infraestructura.** Este subproceso recibe como entrada la salida del subproceso **Analizar la perspectiva de Dominio**. Durante el subproceso se separan los microservicios que tienen procesos funcio-

nales con diferencias de almacenamiento o tráfico en la red. De manera que la granularidad de los microservicios puede disminuir o mantenerse igual pero no aumentar, y el número de microservicios puede aumentar o mantenerse igual, pero no disminuir.

3. **Analizar la perspectiva de Seguridad.** Este subproceso recibe como entrada la salida del subproceso **Analizar la perspectiva de Dominio**. Durante el subproceso se separan los microservicios que tienen procesos funcionales con diferente contextualidad de seguridad o diferente criticalidad de seguridad. De manera que la granularidad de los microservicios puede disminuir o mantenerse igual pero no aumentar, y el número de microservicios puede aumentar o mantenerse igual, pero no disminuir.

4. **Analizar perspectiva de Calidad.** Este subproceso recibe como entrada la salida del subproceso **Analizar la perspectiva de Dominio**. Durante el subproceso se separan los microservicios que tienen procesos funcionales con diferente volatilidad estructural. De manera que la granularidad de los microservicios puede disminuir o mantenerse igual pero no aumentar, y el número de microservicios puede aumentar o mantenerse igual, pero no disminuir.

5. **Resolver contradicciones y Persistencia.** Este subproceso recibe las salidas de los subprocesos: **Analizar la perspectiva de Infraestructura**, **Analizar la perspectiva de Seguridad**, **Analizar la perspectiva de Calidad**. Estas salidas, la mayoría de las veces, tienen sus diferencias. Estas diferencias son llamadas contradicciones. Durante este subproceso el arquitecto de software resuelve las contradicciones con la ayuda del método que se propone. También, durante este subproceso, el arquitecto de software define, para cada microservicios, qué grupos de datos se van a persistir.

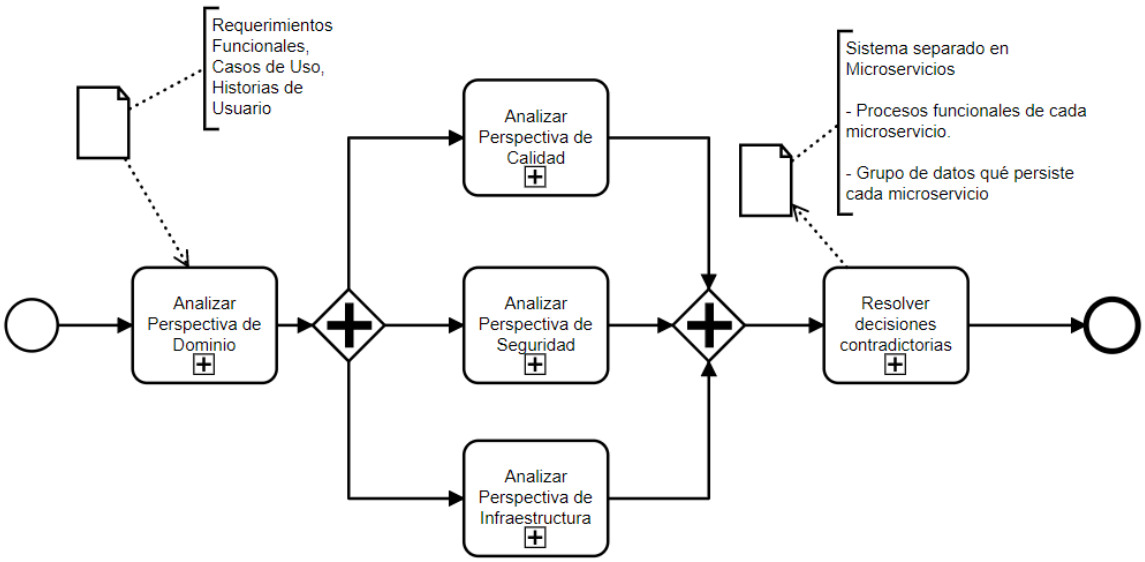

Figura 4-1: Diagrama BPMN de la vista general del método DISC.

4.2. Analizar la perspectiva de dominio

En la figura 4-1, se puede observar un subproceso colapsado llamado "Analizar perspectiva de Dominio". En la figura 4-2, se observa el subproceso expandido junto con las actividades que se realizan durante este subproceso.

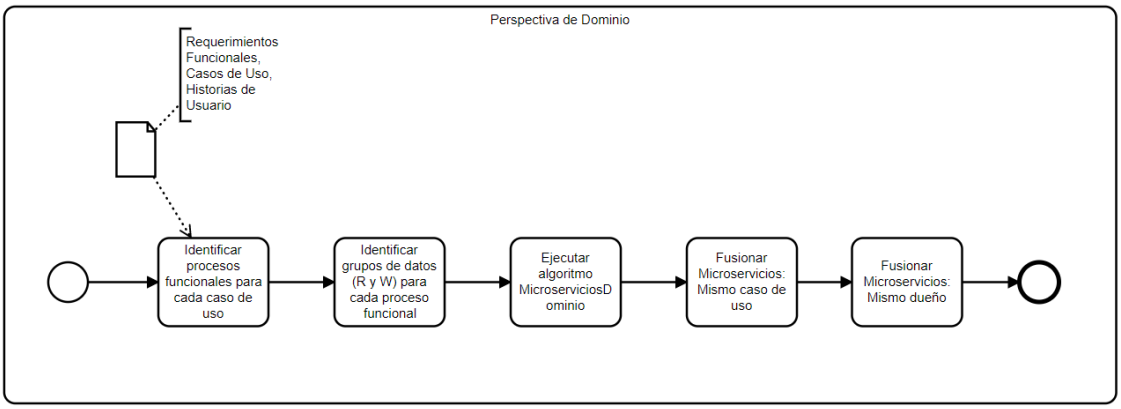

Figura 4-2: Subproceso "Analizar perspectiva de Dominio" expandido.

Este subproceso comienza con la mayor cantidad de microservicios posibles para el método DISC. Cada uno de estos microservicios tiene un valor de granularidad mínimo (1 proceso funcional por microservicio). Conforme avanza el subproceso, la granularidad de los microservicios va aumentando, es decir, el número de procesos

funcionales se incrementa; y el número de microservicios va disminuyendo. Las decisiones de cómo aumenta la granularidad de los microservicios se toman conforme a la perspectiva de dominio.

4.2.1. Identificar procesos funcionales

En la sección 2.4 de esta tesis, se explica el concepto de proceso funcional. Para entender cómo se identifican los procesos funcionales se recomienda leer la sección 3.2 del Manual de Medición COSMIC v4.0.2 [5]. Durante esta actividad, se deben detallar los requerimientos funcionales del usuario para que sean conocidos a un nivel de granularidad donde sus procesos funcionales puedan ser identificados (El concepto de nivel de granularidad de los requerimientos funcionales no tiene nada que ver con el concepto de granularidad de los microservicios).

4.2.2. Identificar relaciones de lectura y escritura

Probablemente en un proyecto de software se utilicen casos de uso o historias de usuario para describir las actividades que debe realizar el sistema. Es importante saber que de un caso de uso se pueden obtener varios procesos funcionales. Un proceso funcional puede mover varios grupos de datos. De un grupo de datos, se pueden obtener varios POJOS (Plain Old Java Objects) de persistencia, o su equivalente en otro lenguaje.

Para el método DISC es necesario encontrar la relación que existe entre los grupos de datos y los procesos funcionales. Es decir, si un proceso funcional *PF1* lee un grupo de datos *GD1* entonces se dice que hay una relación de lectura entre *PF1* y *GD1*. Si *PF1* escribe el grupo de datos *GD2*, entonces se dice que hay una relación de escritura entre *PF1* y *GD2*.

La finalidad de esta actividad es conocer explicitamente las relaciones de lectura y escritura que existen entre todos los procesos funcionales y todos los grupos de datos. La forma más sencilla de lograr esto es mediante una hoja de cálculo (Ver figura 4-3 para un ejemplo).

Procesos Funcionales\Grupos de datos	GD1	GD2	GD3	GD4	GD5
PF1	W	R	W		
PF2	R			R	
PF3	R	W			R
PF4	W		R		
PF5	R			R	W
PF6		R	W		
PF7		R		W	

Figura 4-3: Relaciones entre procesos funcionales y grupos de datos. W para relación de escritura y R para relación de lectura.

4.2.3. Ejecutar algoritmo MicroserviciosDominio

Conociendo las relaciones entre procesos funcionales y grupos de datos, se puede ejecutar el siguiente algoritmo llamado "MicroserviciosDominio".

```
/**
* Este algoritmo busca agregar a un mismo microservicio
* todos los procesos funcionales que escriban un mismo
* grupos de datos.
*/
MicroserviciosDominio(GDs, PFs, Rs){
    //Se crea un microservicio por cada Proceso Funcional
    Ms <- crearMicroservicios(PFs)
    foreach GD in GruposDatos
        //Método que regresa una lista de todos los
        //microservicios que contienen procesos
        //funcionales con relación de escritura con GD
        MsGd <- obtenEscritura(Rs, Ms, GD)
        //Método que fusiona todos
        //los Microservicios de una lista
        M <- merge (MsGd)
        //Eliminar de Ms los valores de la lista M
        Ms.eliminar(MsGd)
        //Agregar el nuevo microservicio a la lista
```

```
            Ms.agregar(M)
        }
```

Utilizando este algoritmo, lo que se logra es tener al inicio, un conjunto de microservicios pequeños. Cada microservicio se encarga de un solo proceso funcional. Dos o más microservicios se fusionan si escriben el mismo grupo de datos. Desde el punto de vista de consistencia de datos es mejor que un microservicio escriba en su propia base de datos, en vez de comunicarse por red con otro microservicio para que ese otro microservicio guarde los datos.

Durante esta tarea, la granularidad de los microservicios aumenta y el número de microservicios disminuye. El aumento de la granularidad se da tomando en cuenta las relaciones de escritura entre los procesos funcionales y los grupos de datos.

4.2.4. Fusionar Microservicios: Mismo caso de uso

En esta actividad, se fusionan los microservicios que contienen procesos funcionales que pertenecen al mismo caso de uso (o historia de usuario). En la sección 2.1.3 de esta tesis, se mencionan criterios de acoplamiento para microservicios. Uno de ellos es **CC-2 Proximidad Semántica**. Este busca que las nanoentidades que son semánticamente cercanas pertenezcan a un mismo microservicio. Por ejemplo, cuando dos nanoentidades pertencen a un mismo caso de uso. El método DISC no utiliza el concepto de nanoentidad, si no el de proceso funcional. Sin embargo, la idea de proximidad semántica sigue siendo la misma (procesos funcionales que pertenecen a un mismo caso de uso/historia de usuario).

Durante esta tarea, la granularidad de los microservicios aumenta. El aumento de la granularidad se da al fusionar los microservicios que tienen procesos funcionales que pertenecen al mismo caso de uso.

4.2.5. Fusionar Microservicios: Mismo dueño

En esta actividad, se fusionan los microservicios que contienen procesos funcionales que tienen un mismo dueño. Existe un criterio de acoplamiento llamado **CC-3**

Dueño Compartido que busca que las nanoentidades que sean responsabilidad de un mismo dueño pertenezcan a un mismo servicio. El dueño se puede definir como el usuario funcional del proceso funcional, o también puede ser un rol o un departamento. Esta actividad se puede ignorar si el arquitecto de software lo considera necesario. Por ejemplo, si todos los procesos funcionales tienen un mismo dueño, entonces esta actividad se debería evitar con el fin de no fusionar todos los microservicios en uno solo.

Al realizar esta tarea, la granularidad de los microservicios tiende a aumentar. El aumento se da al fusionar los microservicios que tengan procesos funcionales pertenecientes a un mismo dueño.

4.2.6. Ejemplo

Durante todo el capítulo 4, se sigue un ejemplo que permite observar cómo hacer uso de este método. Para evitar confusiones, se intenta mantener el ejemplo lo más simple posible. El ejemplo no está basado en un sistema real. Su única función es observar las entradas y salidas de cada actividad, con el fin de que se comprenda mejor el método DISC.

Se supone que durante la actividad **"Identificar procesos funcionales para cada caso de uso"** se encontraron, en 4 casos de uso, 14 procesos funcionales. En la figura 4-4 se observa qué procesos funcionales pertenecen a qué casos de uso.

Caso de Uso 1
PF1
PF2
PF3
PF4

Caso de Uso 2
PF5
PF6
PF7

Caso de Uso 3
PF8
PF9
PF10
PF11

Caso de Uso 4
PF12
PF13
PF14

Figura 4-4: Ejemplo de la salida de la actividad **"Identificar procesos funcionales para cada caso de uso"**.

Habiendo identificado los procesos funcionales, se debe realizar la actividad **"Identificar relaciones de lectura y escritura"**. Se supondrá que un desarrollador o arquitecto de software que conoce COSMIC realiza una revisión de los requerimientos

funcionales y llega a la conclusión de que las relaciones de lectura y escritura para este sistema son las que se muestran en la figura 4-5.

Procesos Funcionales/Grupos de datos	GD1	GD2	GD3	GD4
PF1	W		R	
PF2	R			
PF3	R			R
PF4	W			
PF5	W	W		R
PF6		R		
PF7		R		
PF8	R		W	
PF9		R	W	
PF10			R	
PF11			R	
PF12				R
PF13			R	W
PF14				R

Figura 4-5: Ejemplo de la salida de la actividad **"Identificar relaciones de lectura y escritura"**

Con esta información, se puede realizar la siguiente actividad **"Ejecutar algoritmo MicroserviciosDominio"**. La ejecución de este algoritmo tiene como salida un conjunto de microservicios con ciertos procesos funcionales pertenecientes a cada microservicio. Con la información que se tiene, la salida del algoritmo son 11 microservicios que se muestran en la figura 4-6.

Teniendo estos 11 microservicios, lo siguiente que se hace es la actividad **"Fusionar Microservicios: Mismo caso de uso"**. Ya se conoce cuales son los procesos funcionales que pertenecen a cada caso de uso (4-4), por lo que se puede comenzar a realizar la actividad. El proceso de esta actividad se puede observar en la figuras 4-7, 4-8, 4-9 en ese orden. La salida de esta actividad para el ejemplo se puede observar en la figura 4-10.

La última actividad de este subproceso es **"Fusionar Microservicios: Mismo dueño"**. Se debe definir con antelación el significado que se le da a la palabra dueño (una persona, un tipo de usuario, un departamente, etc). En este caso, se piensa que el dueño es un tipo de usuario. Se va a suponer que los procesos funcionales del Caso de

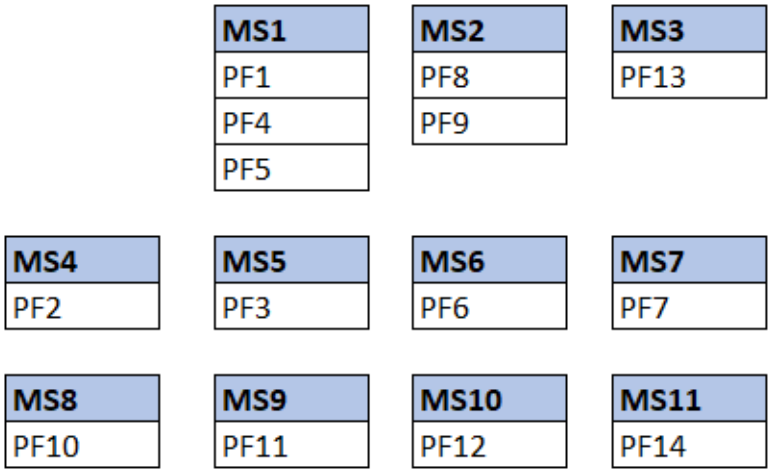

Figura 4-6: Ejemplo de la salida de la actividad **"Ejecutar algoritmo MicroserviciosDominio"**.

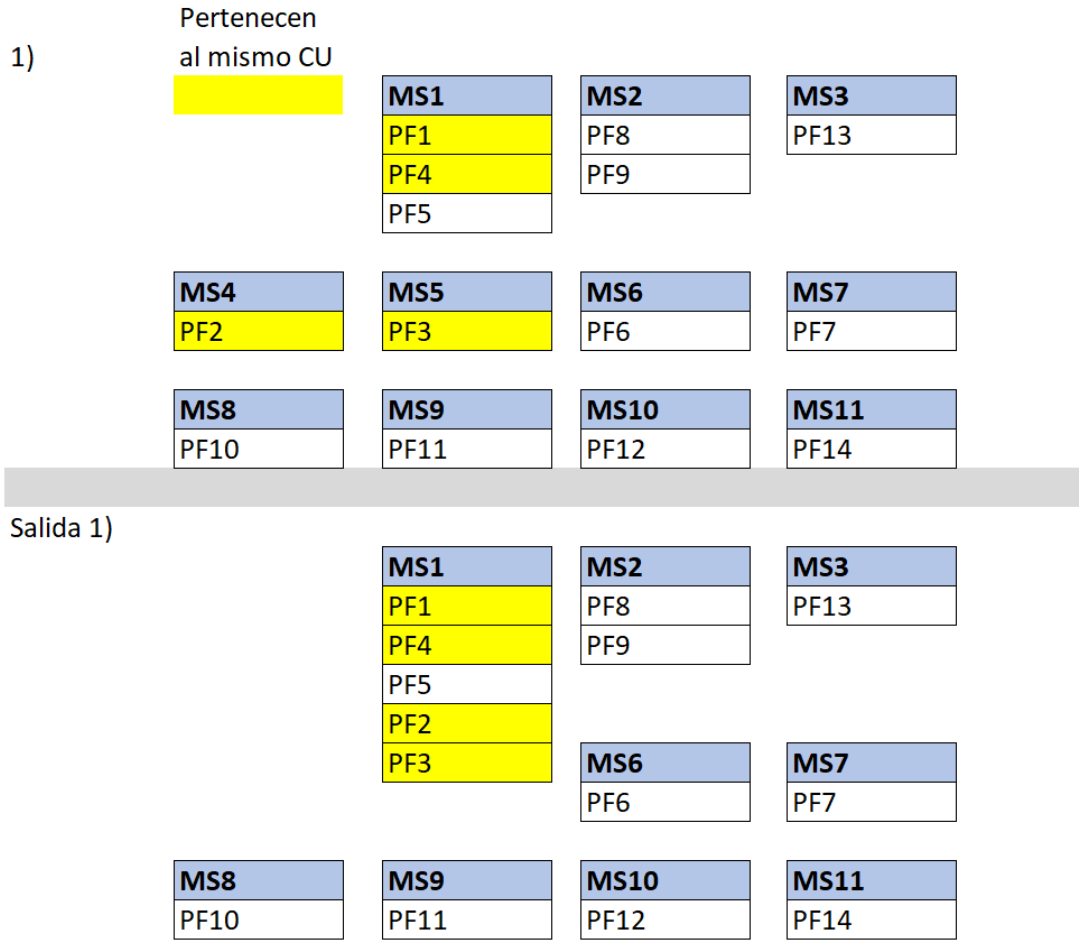

Figura 4-7: Proceso (paso 1) de la actividad **"Fusionar Microservicios: Mismo caso de uso"**

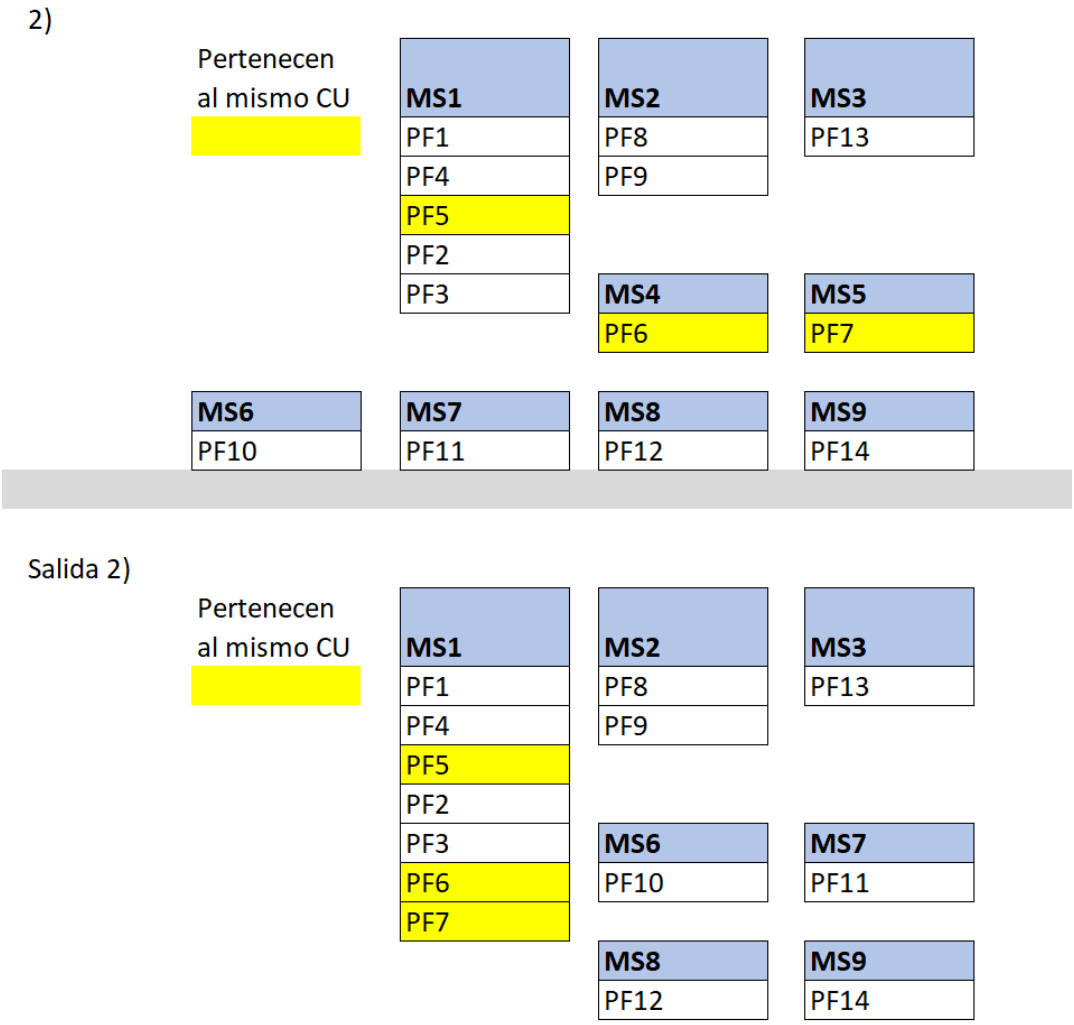

Figura 4-8: Proceso (paso 2) de la actividad **"Fusionar Microservicios: Mismo caso de uso"**

Uso 1 pertenecen a un usuario tipo "admin", todos los procesos funcionales del Caso de Uso 2 pertencen a un usuario tipo "empleado" y todos los procesos funcionales del Caso de Uso 3 y 4 pertenecen a un usuario tipo "cliente". Con estas suposiciones, el resultado de esta actividad se muestra en la figura 4-11.

4.3. Analizar la perspectiva de Infraestructura

En la figura 4-1, se puede observar un subproceso colapsado llamado "Analizar perspectiva de Infraestructura". En la figura 4-12, se observa el subproceso expandido

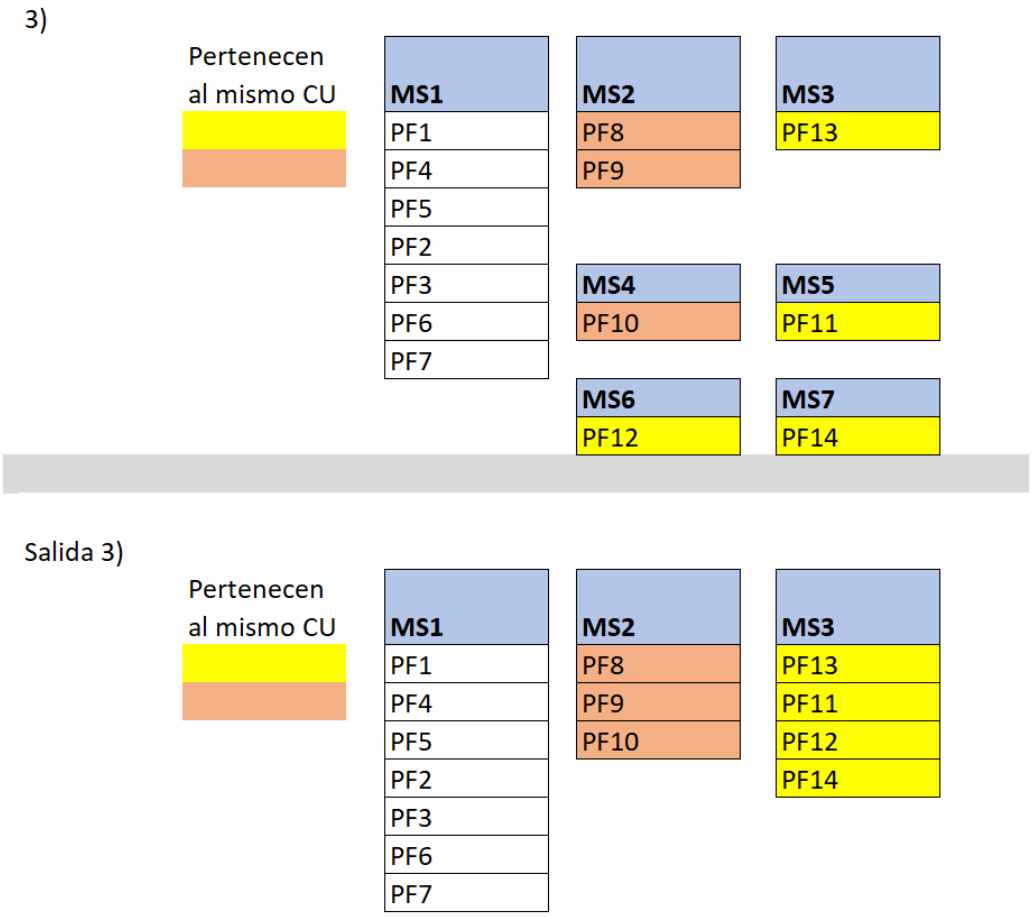

Figura 4-9: Proceso (paso 3) de la actividad **"Fusionar Microservicios: Mismo caso de uso"**

MS1	MS2	MS3
PF1	PF8	PF11
PF2	PF9	PF12
PF3	PF10	PF13
PF4		PF14
PF5		
PF6		
PF7		

Figura 4-10: Ejemplo de la salida de la actividad **"Fusionar Microservicios: Mismo caso de uso"**

junto con las actividades que se realizan durante este subproceso.

Este subproceso recibe como entrada la salida del subproceso **Analizar perspectiva de Dominio**. Este subproceso toma los microservicios que recibe como entrada

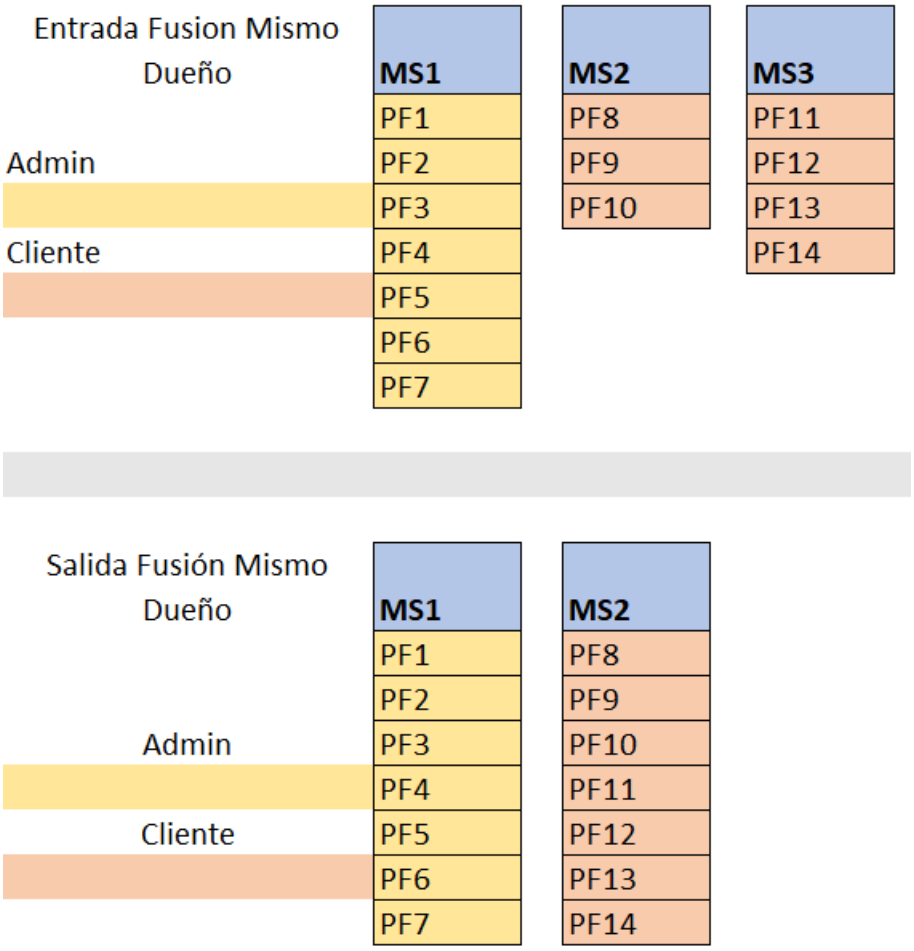

Figura 4-11: Ejemplo de la salida de la actividad **"Fusionar Microservicios: Mismo dueño"**.

y, de ser necesario, los separa. De manera que lo que este subproceso hace es disminuir la granularidad de microservicios y aumentar el número de microservicios. Las decisiones se basan en la perspectiva de infraestructura.

4.3.1. Separar Microservicios: Diferencia de Almacenamiento

En esta actividad, se separan los microservicios que contienen procesos funcionales que no son semejantes en cuanto a su almacenamiento. En la sección 2.1.3 de esta tesis se presenta un criterio de acoplamiento llamado **CC-11 Semejanza de Almacenamiento**. En éste, se menciona que las nanoentidades que guardan archivos con

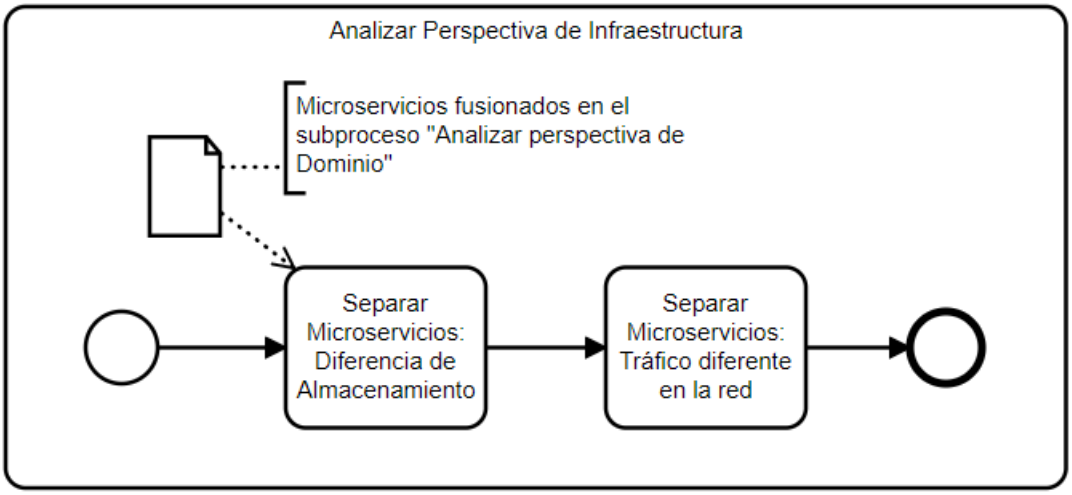

Figura 4-12: Subproceso “Analizar perspectiva de Infraestructura” expandido.

una diferencia de tamaño significativa no deberían pertenecer al mismo microservicio. La granularidad de los microservicios disminuye porque los microservicios se separan conforme a las diferencias de almacenamiento que se encuentran.

Durante esta actividad se clasifican los niveles de almacenamiento de todos los procesos funcionales. Ésto se hace para cada microservicio obtenido del subproceso “Analizar Perspectiva de Dominio”. Los niveles de clasificación dependen del sistema y del arquitecto de software. Para algunos sistemas, 1 MB se puede clasificar como alto nivel de almacenamiento, y para otros sistemas 1GB se podría clasificar como un nivel de almacenamiento regular. Habiendo clasificado los procesos funcionales, se pasa a separar los microservicios segun los niveles de clasificación de los procesos funcionales. Se piensa que la forma más sencilla de realizar esta actividad es usando una hoja de cálculo. Se puede observar un ejemplo en la figura 4-13.

	MS1					
	Caso de Uso 1			Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6
Nivel de Almacenamiento	Alto	Alto	Bajo	Bajo	Regular	Regular

Figura 4-13: Clasificando procesos funcionales conforme a su nivel de almacenamiento.

Con el ejemplo de la figura 4-13, se puede pensar que se tienen que hacer un nuevo microservicio con PF1 y PF2. Sin embargo, esto rompería con el criterio de

acoplamiento **CC-2 Proximidad Semántica**, ya que PF3 pertenece al mismo caso de uso que PF1 y PF2. En este caso, se recomienda hacer un microservicio con PF1, PF2 y PF3, con el fin de mantener la proximidad semántica y cumplir lo mejor posible el criterio de **CC-11 Semejanza de Almacenamiento**.

4.3.2. Separar Microservicios: Tráfico diferente en la red

En esta actividad, se separan los microservicios que tienen procesos funcionales con tráfico diferente en la red. En la sección 2.1.3 de esta tesis se presenta un criterio de acoplamiento llamado **CC-13 Tráfico Similar en la Red**. La idea de este criterio de acoplamiento es crear microservicios pequeños con nanoentidades pequeñas y poco frecuentadas. Al ser poco frecuentadas, el tráfico de la red es poco afectado. Durante esta tarea la granularidad de los microservicios tiende a disminuir. Ésto porque los microservicios se van separando conforme a las diferencias de tráfico en la red que se encuentran.

Al igual que la actividad pasada, se clasifican los niveles de tráfico en la red para cada microservicio obtenido de la actividad "Separar Microservicios: Diferencia de Almacenamiento". Despues de clasificados los procesos funcionales, se pasa a separar los microservicios conforme a las diferencias encontradas en tráfico en la red. Se utiliza una hoja de cálculo para realizar esta actividad. Se puede observar un ejemplo en la figura 4-14.

	MS1					
	Caso de Uso 1			Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6
Tráfico en la red	Bajo	Bajo	Regular	Regular	Alto	Regular

Figura 4-14: Clasificando procesos funcionales conforme a su tráfico en la red.

El ejemplo de la figura 4-14 puede llevar a pensar que se debe crear otro microservicio con PF1 y PF2. Sin embargo, esto rompería con el criterio de acoplamiento **CC-2 Proximidad Semántica**, ya que PF3 pertenece al mismo caso de uso que PF1 y PF2. En este caso se recomienda crear un microservicio aparte con PF1, PF2 y PF3, o simplemente dejar el microservicio como está.

4.3.3. Ejemplo

Siguiendo con el ejemplo de la sección 4.2.6, se realizan las actividades del subproceso **Analizar perspectiva de Infraestructura**. Se supone que durante la actividad un analista de software realiza una clasificación del nivel de almacenamiento requerido para cada proceso funcional, obteniendo los resultados de la figura 4-15. Ahora el arquitecto de software debe realizar la actividad **Separar Microservicios: Diferencia de Almacenamiento** tomando en cuenta la clasificación de nivel de almacenamiento que realiza realiza el analista.

	Microservicio 1						
	Caso de Uso 1				Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6	PF7
Nivel de Almacenamiento	Alto	Alto	Alto	Bajo	Bajo	Regular	Bajo

	Microservicio 2						
	Caso de Uso 3				Caso de Uso 4		
	PF8	PF9	PF10	PF11	PF12	PF13	PF14
Nivel de Almacenamiento	Bajo	Regular	Regular	Bajo	Bajo	Regular	Bajo

Figura 4-15: Clasificando procesos funcionales conforme a su nivel de almacenamiento.

Los procesos funcionales PF1, PF2 y PF3 tienen un nivel de almacenamiento alto, mientras que PF5, PF6 y PF7 tienen almacenamiento regular o bajo. Por lo que se decide separar el microservicio 1 en 2 microservicios. Para el microservicio 2, los niveles de almacenamiento de los procesos funcionales oscilan entre regular y bajo, el arquitecto puede tomar la decisión de separar o no este microservicio. Para el caso del ejemplo se supone que el arquitecto decide no separar el microservicio. La salida de esta actividad se puede observar en la figura 4-16.

Siguiendo con el subproceso, se realiza la actividad **Separar Microservicios: Tráfico diferente en la red**. La entrada a esta actividad es la salida de la actividad pasada (Figura 4-16). Se supone que un analista de software realiza una clasificación de cada proceso funcional conforme a su nivel de tráfico en la red. Se pueden observar los resultados de esta clasificación en la figura 4-17. Tomando en cuenta la clasificación el arquitecto de Software debe tomar decisiones sobre separar o no los microservicios.

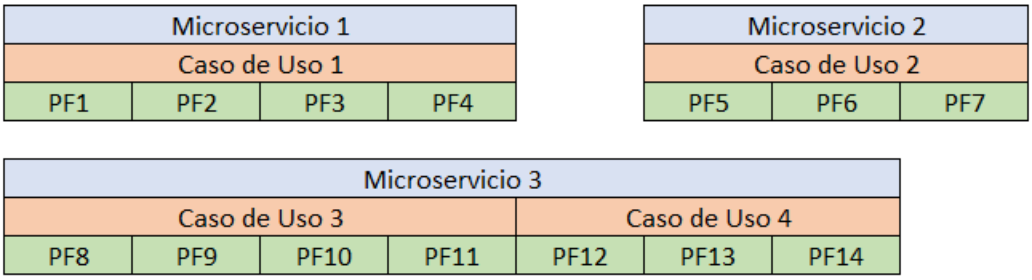

Figura 4-16: Salida de la actividad **Separar Microservicios: Diferencia de Almacenamiento**.

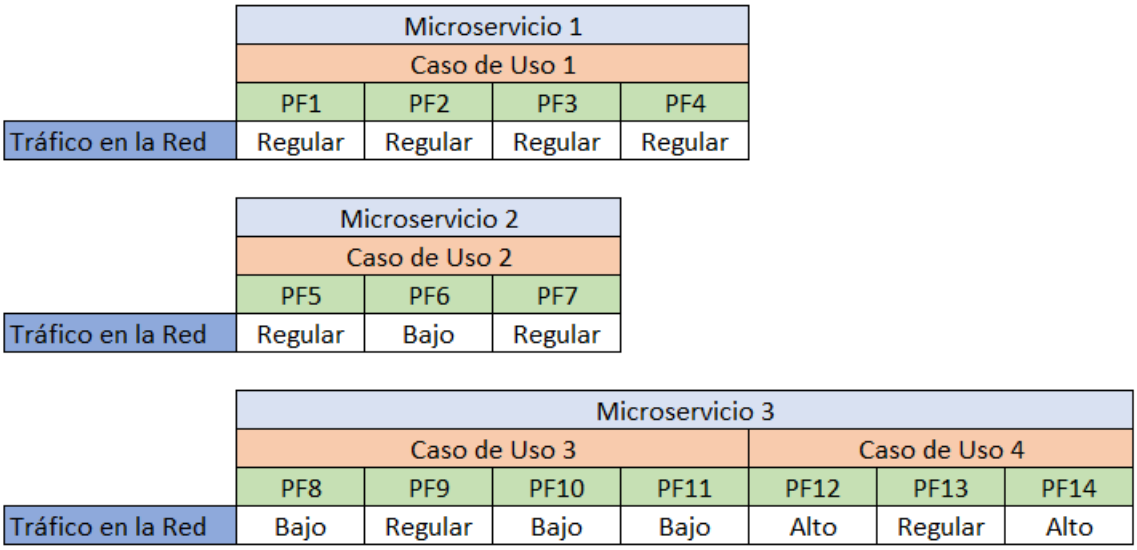

Figura 4-17: Clasificando microservicios conforme a su tráfico en la red.

El arquitecto decide no separar los Microservicio 1 y 2, por 2 razones: Se dejaría de cumplir el criterio **CC-2 Proximidad Semántica**, y los niveles de tráfico en la red no son tan variantes dentro de un mismo microservicio. En cambio el Microservicio 3 sí puede ser separado. Tres de los cuatro procesos funcionales del Caso de Uso 3 cuentan con un nivel de tráfico en la red bajo. Por lo que se toma la decisión de separar estos 4 procesos funcionales para crear otro microservicio. La salida de esta actividad se puede observar en la figura 4-18.

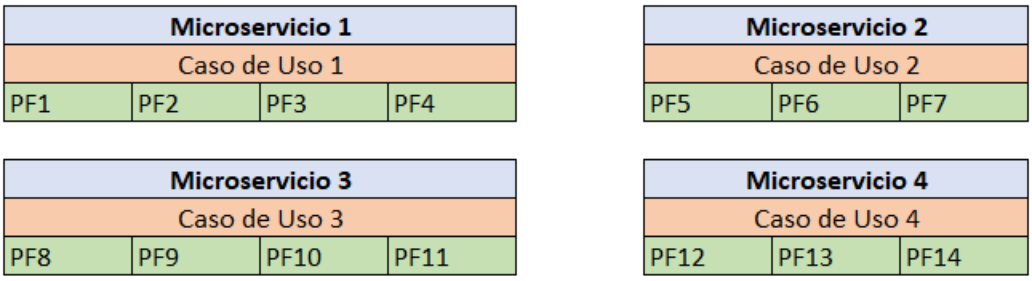

Figura 4-18: Salida de la actividad **Separar Microservicios: Tráfico diferente en la red**.

4.4. Analizar perspectiva de Seguridad

En la figura 4-1 se puede observar un subproceso colapsado llamado "Analizar perspectiva de Seguridad". En la figura 4-19 se observa el subproceso expandido junto con las actividades que se realizan durante este subproceso.

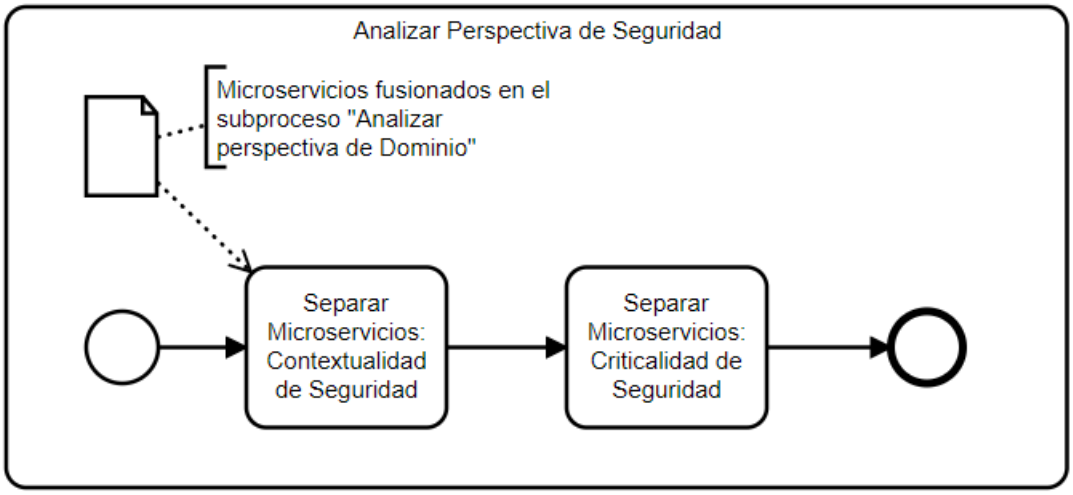

Figura 4-19: Subproceso "Analizar perspectiva de Seguridad" expandido.

Este subproceso recibe como entrada la salida del subproceso **Analizar perspectiva de Dominio**. Este subproceso toma los microservicios que recibe como entrada y, de ser necesario, los separa. De manera que, lo que este subproceso hace es disminuir la granularidad de microservicios y aumentar el número de microservicios. Las decisiones se basan en la perspectiva de seguridad.

4.4.1. Separar Microservicios: Criticalidad de Seguridad

Durante esta actividad, se separan los microservicios que tienen diferentes niveles de criticalidad de la Seguridad basandose en el criterio de acoplamiento **CC-15 Criticalidad de Seguridad**. Un nivel de seguridad alto tiene un costo más alto de desarrollo (se necesita un equipo de desarrollo con experiencia en seguridad) y de procesamiento. Durante esta actividad, la granularidad de los microservicios tiende a disminuir. Ésto porque los microservicios se separan si sus procesos funcionales tienen diferentes niveles de criticalidad de seguridad.

Al realizar esta actividad, se deben clasificar los procesos funcionales conforme a su nivel de criticalidad de seguridad. En este caso, se utilizan 2 valores para clasificar la seguridad: regular y alto. Esto se hace para cada microservicio que se obtuvo del subproceso "Analizar Perspectiva de Dominio". Habiendo realizado la clasificación, se pasa a separar los microservicios que tienen procesos funcionales con diferencias altas de criticalidad de seguridad. Se utiliza una hoja de cálculo para realizar esta actividad. Se puede observar un ejemplo en la figura 4-20.

	MS1					
	Caso de Uso 1			Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6
Criticalidad de Seguridad	Alto	Regular	Regular	Regular	Regular	Regular

Figura 4-20: Clasificando procesos funcionales conforme a su criticalidad de seguridad.

El ejemplo de la figura 4-20 puede llevar a pensar que se debe crear un nuevo microservicio solo con PF1, sin embargo esto rompería con el criterio de acoplamiento **CC-2 Proximidad Semántica**. Para este caso, se piensa que la mejor decisión es generar un nuevo microservicio con PF1, PF2 y PF3. De esta manera, un equipo de desarrollo con experiencia en seguridad se puede encargar del desarrollo y la implementación de este microservicio. Sin embargo, otro arquitecto de software puede pensar diferente.

4.4.2. Separar Microservicios: Contextualidad de Seguridad

Durante esta actividad, se separan los microservicios que contienen diferentes contextos de seguridad. En la sección 2.1.3 se presenta un criterio de acoplamiento llamado **CC-14 Contextualidad de Seguridad**. Este criterio de acoplamiento se parece a **CC-3 Dueño compartido**. Sin embargo, el criterio de acoplamiento **CC-14 Contextualidad de Seguridad** agrega que mezclar contextos de seguridad en un servicio (diferentes roles de seguridad) complica implementar la autenticación y autorización del sistema.

Lo que se realiza durante esta actividad es clasificar qué roles acceden a qué procesos funcionales. En este caso uno o más roles de seguridad pueden acceder a uno o más procesos funcionales. Una manera sencilla de observar esta relación proceso funcional-rol es como un grafo. Se muestra un ejemplo en la figura 4-21.

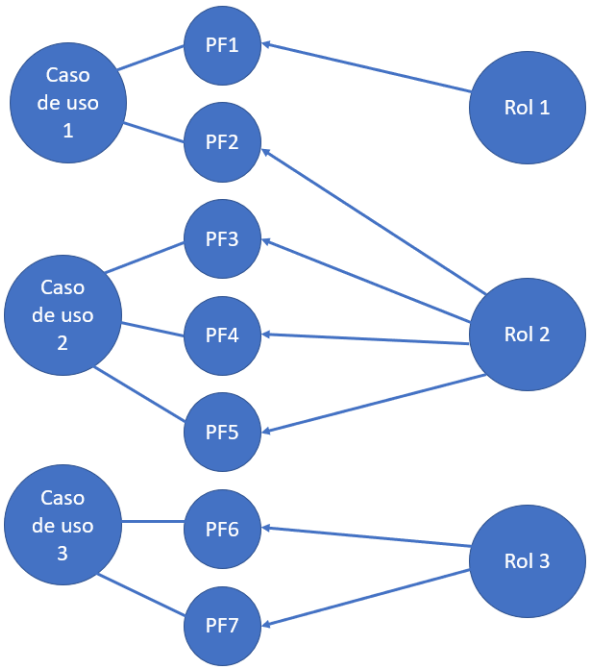

Figura 4-21: Relación proceso funcional-rol para un microservicio.

En la figura 4-22, se muestra un ejemplo de como se separaría, de forma incorrecta, el microservicio representado en la figura 4-21.

El criterio de acoplamiento **CC-14 Contextualidad de Seguridad** menciona que no se deben mezclar diferentes contextos de seguridad porque esto complica la

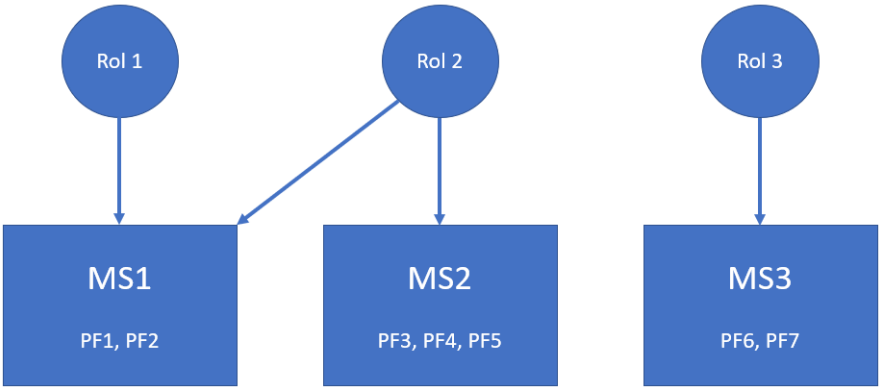

Figura 4-22: Separación incorrecta del microservicio representado en la figura 4-21.

implementación de la autorización y autenticación. Es decir, uno de los objetivos de este criterio de acoplamiento es evitar complicar la autenticación y autorización. Si se separa el sistema como se muestra en la figura 4-22 los microservicios MS1 y MS2 tendrían que implementar la autorización y autenticación del Rol 2. Es decir, no se evita la complicación de la autorización y autenticación para ni uno de los 2 microservicios.

La manera correcta de separar el microservicio representado en la figura 4-21, se muestra en la figura 4-23. Se puede observar que MS1 de la figura 4-22 maneja los mismos roles que MS1 de la figura 4-23.

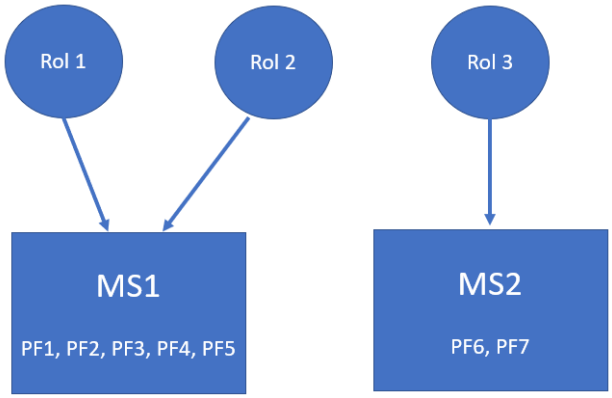

Figura 4-23: Separación correcta del microservicio representado en la figura 4-21.

La manera más sencilla de realizar esta actividad es representar las relaciones proceso funcional-rol como un grafo no dirigido. La figura 4-25 muestra como se representa el ejemplo de la figura 4-24 adaptada a un grafo G no dirigido. Si dentro

de G se encuentran dos o más subgrafos disconexos entonces cada uno de estos grafos disconexos representan un microservicio. Si G es conexo entonces, no se realiza ningun cambio durante esta actividad.

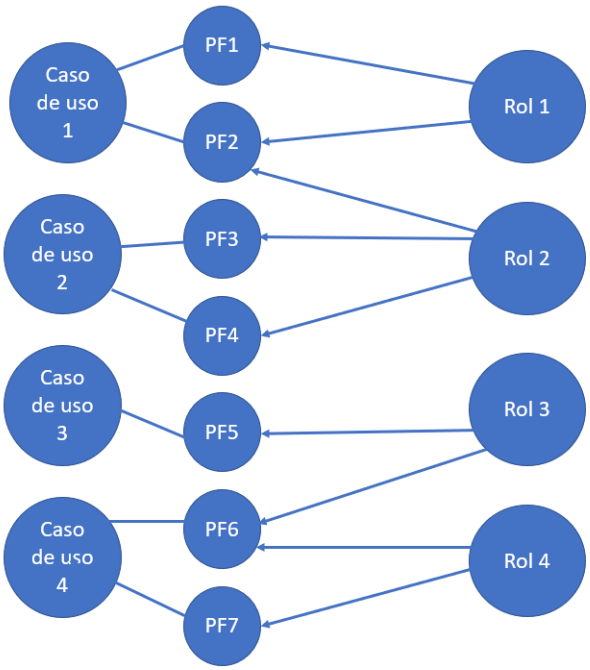

Figura 4-24: Relación proceso funcional-rol para un microservicio.

En la figura 4-25, se pueden observar 2 subgrafos disconexos. La salida de esta actividad se observa en la figura 4-26.

Durante esta actividad, la granularidad de los microservicios tiende a disminuir. Esto sucede porque se separan los microservicios que contienen procesos funcionales con diferentes contextos de seguridad.

4.4.3. Ejemplo

Siguiendo con el ejemplo de la sección 4.2.6 se realizan las actividades del subproceso **Analizar perspectiva de Seguridad**.

Para este ejemplo se supone que un experto en seguridad de sistemas realiza una clasificación de la Criticalidad de Seguridad para cada uno de los procesos funcionales. Los resultados de esta clasificación se muestran en la figura 4-27. Ahora, el arquitecto de software puede realizar la tarea **Separar Microservicios: Criticali-**

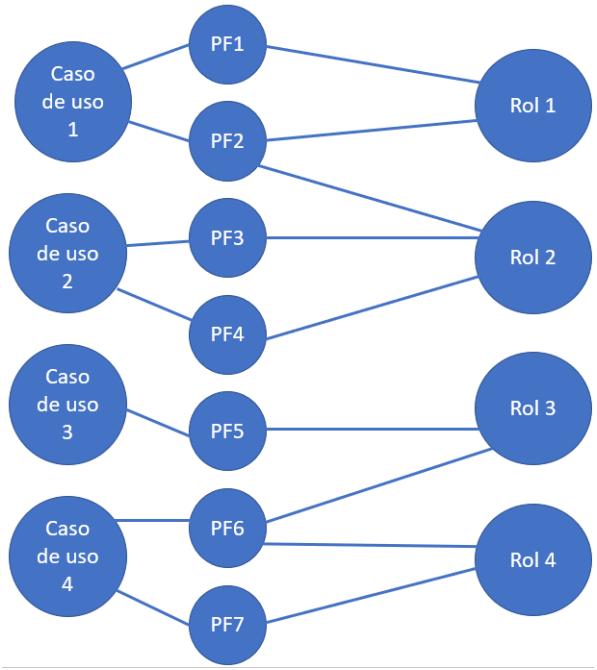

Figura 4-25: Relaciones proceso funcional-rol de la figura 4-24 representado como un grafo no dirigido.

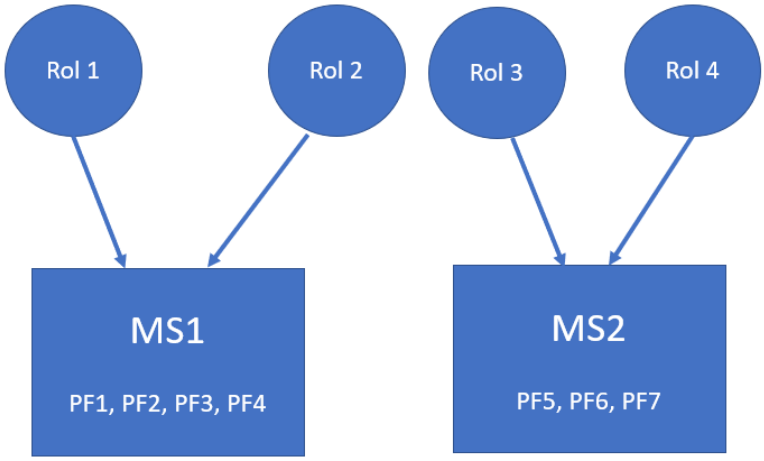

Figura 4-26: Separación correcta del microservicio representado en la figura 4-24.

dad de Seguridad tomando en cuenta la información proporcionada por el experto en seguridad.

El arquitecto decide que MS1 se va a separar en 2 microservicios. Esto porque varios procesos funcionales del Caso de Uso 1 tienen una criticalidad de seguridad alta; y ni un proceso funcional del Caso de Uso 2 tiene criticalidad alta. Para el caso de MS2, el arquitecto de software observa que hay procesos funcionales del Caso

	MS1						
	Caso de Uso 1				Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6	PF7
Criticalidad de Seguridad	Alto	Alto	Regular	Regular	Regular	Regular	Regular

	MS2						
	Caso de Uso 3				Caso de Uso 4		
	PF8	PF9	PF10	PF11	PF12	PF13	PF14
Criticalidad de Seguridad	Regular	Alto	Regular	Alto	Alto	Regular	Regular

Figura 4-27: Clasificando procesos funcionales conforme a su criticalidad de seguridad.

de Uso 3 y del Caso de Uso 4 que tienen criticalidad de seguridad alta. Por esto se considera que no es necesario separar MS2. Un mismo equipo de desarrollo con experiencia en seguridad puede encargarse de éste último microservicio. Al culminar este actividad, el arquitecto presenta los resultados que se muestran en la figura 4-28.

MS1			
Caso de Uso 1			
PF1	PF2	PF3	PF4

MS2		
Caso de Uso 2		
PF5	PF6	PF7

MS3						
Caso de Uso 3				Caso de Uso 4		
PF8	PF9	PF10	PF11	PF12	PF13	PF14

Figura 4-28: Salida de la actividad **Separar Microservicios: Criticalidad de Seguridad**.

Posteriormente, el arquitecto debe realizar la actividad **Separar Microservicios: Contextualidad de Seguridad**. Como se menciona en la sección anterior, se crea un grafo no dirigido por cada microservicio. Este grafo muestra las relaciones que existen entre procesos funcionales y roles dentro de un mismo microservicio.

En la sección 4.2.6 de esta tesis, se menciona que todos los procesos funcionales del Caso de Uso 1 pertenecen a un usuario tipo “admin”, los procesos funcionales del caso de uso 2 pertenecen a un usuario tipo “empleado”, y los procesos funcionales del Caso de Uso 3 y 4 pertencen a un usuario tipo “cliente”. Con esta información, se

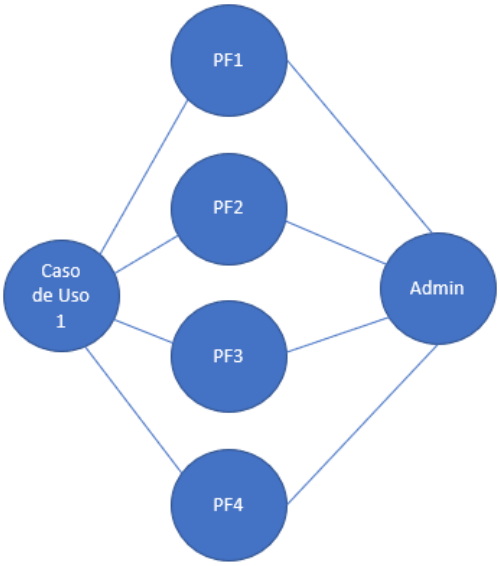

Figura 4-29: Relaciones proceso funcional-rol de MS1 representado como un grafo no dirigido.

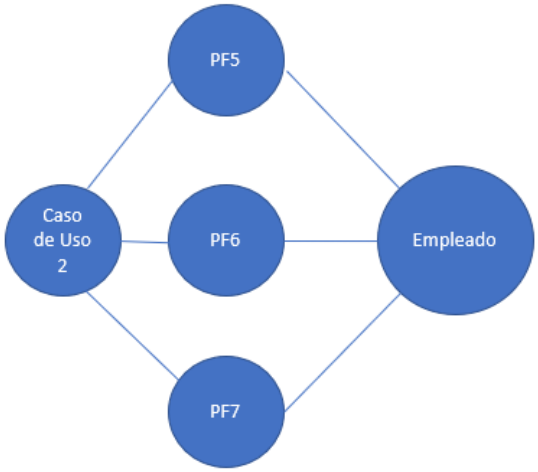

Figura 4-30: Relaciones proceso funcional-rol de MS2 representado como un grafo no dirigido.

generan los grafos de la figuras 4-29, 4-30 y 4-31. Se puede observar que en ninguna de las 3 figuras que no hay subgrafos disconexos, por esto no se generan cambios en los microservicios. Terminando el subproceso los microservicios se representan como se muestra en la figura 4-28.

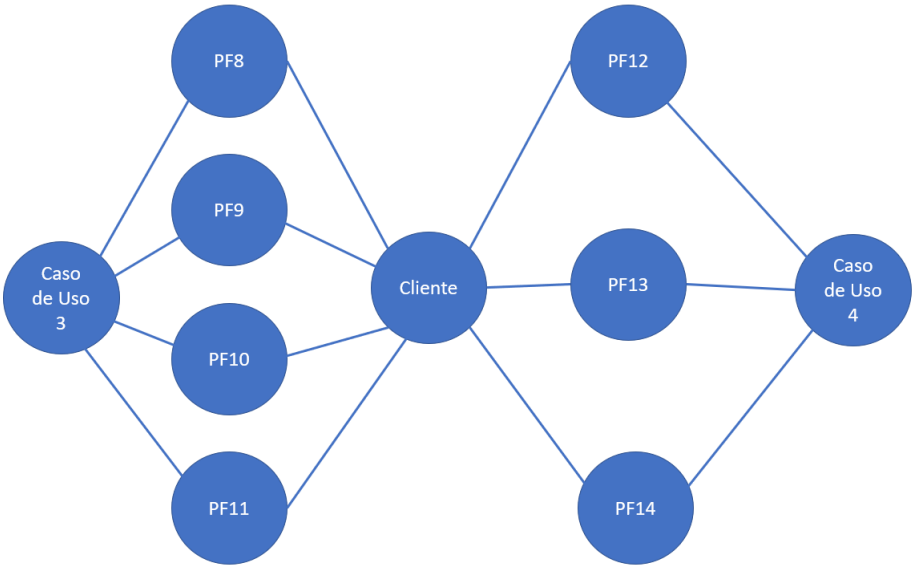

Figura 4-31: Relaciones proceso funcional-rol de MS3 representado como un grafo no dirigido.

4.5. Analizar perspectiva de Calidad

En la figura 4-1 se puede observar un subproceso colapsado llamado “Analizar perspectiva de Calidad”. En la figura 4-32, se observa el subproceso expandido junto con las actividades que se realizan durante este subproceso.

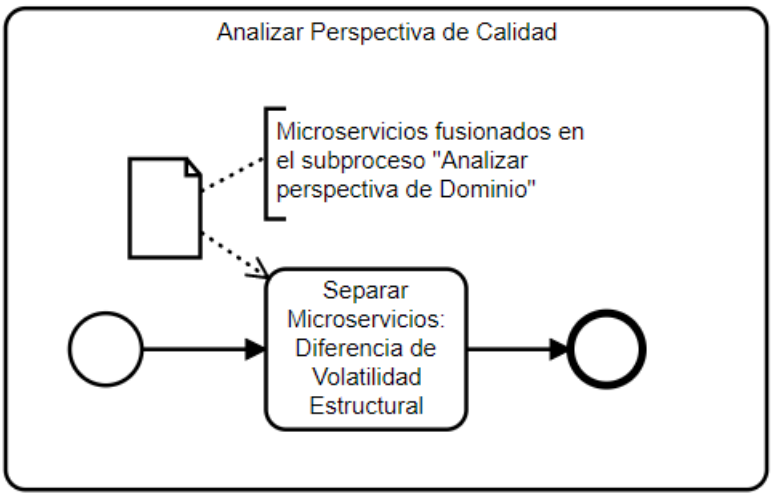

Figura 4-32: Subproceso “Analizar perspectiva de Calidad” expandido.

Al tener solo una actividad, se podría pensar que el método DISC solo toma en cuenta un criterio de acoplamiento de perspectiva de calidad. Sin embargo, esto no es

así: otros criterios de acoplamiento de perspectiva de calidad se utilizan en el próximo subproceso “Resolver Contradicciones”.

Este subproceso, Analizar perspectiva de Calidad, recibe como entrada la salida del subproceso **Analizar perspectiva de Dominio**. El subproceso toma los microservicios que recibe como entrada y, de ser necesario, los separa. De manera que este subproceso disminuye granularidad de microservicios y aumenta el número de microservicios. Las decisiones se basan en la perspectiva de calidad.

4.5.1. Separar Microservicios: Diferencia de Volatilidad Estructural

Durante esta actividad, se separan los procesos funcionales estructuralmente volátiles de los no-estructuralmente volátiles. El criterio de acoplamiento **CC-4 Volatilidad Estructura** menciona que se debe buscar separar las nanoentidades volátiles de las no volátiles. La descomposición de servicios debe ser realizada de una manera que favorezca el compartir las nanoentidades no volátiles porque, cada cambio al código, puede agregar errores al software. El criterio de acoplamiento **CC-4 Volatilidad Estructural** pertenece a la perspectiva de dominio, pero por la naturaleza de que multiples cambios al código al código hacen más propensos los errores y por conveniencia del método se toma en cuenta como perspectiva de calidad.

Un ejemplo de un proceso funcional volátil puede ser cierto proceso funcionales que, en un futuro, vaya a escalar. En caso de no contar con la información sobre futuros cambios que puede tener el sistema, se puede ignorar esta actividad y pasar directo al subproceso “Resolver Contradicciones”.

En esta actividad, se clasifican los procesos funcionales como volátiles o no volátiles. Esto conforme a la información que se conozca sobre como se va a modificar el software en producción. Se puede observar un ejemplo de esta clasificación en la figura 4-33. En esta figura se puede observar que PF1 y PF3 se clasifican como volátiles. Por esto, se decide generar un nuevo microservicio con los procesos funcionales pertenecientes al Caso de Uso 1. La salida de esta actividad se puede observar en la

figura 4-34.

	MS1					
	Caso de Uso 1			Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6
Volatilidad	Volatil	No volatil	Volatil	No Volatil	No Volatil	No Volatil

Figura 4-33: Clasificando procesos funcionales conforme a su volatilidad.

	MS1		
	Caso de Uso 1		
	PF1	PF2	PF3
Volatilidad	Volatil	No volatil	Volatil

	MS2		
	Caso de Uso 2		
	PF4	PF5	PF6
Volatilidad	No Volatil	No Volatil	No Volatil

Figura 4-34: Salida de la actividad "Separar Microservicios: Diferencia de Volatilidad" para ejemplo de la figura 4-33.

Durante esta actividad, la granularidad de los microservicios tiende a disminuir. Esto sucede porque se separan los procesos funcionales no volátiles de los volátiles.

4.5.2. Ejemplo

Siguiendo con el ejemplo de la sección 4.2.6, se realizan las actividades del subproceso **Analizar perspectiva de Calidad**. Para este ejemplo se supone que una persona del equipo de aseguramiento de la calidad realizo una clasificación de cada proceso funcional conforme a su volatilidad. Los resultados de esta clasificación se pueden observar en la figura 4-35. Se supone que durante la actividad **Separar Microservicios: Diferencia de Volatilidad** se clasifican los procesos funcionales como se muestra en la figura .

El arquitecto de software pasa a realizar la tarea **Separar Microservicios: Diferencia de Volatilidad**. El arquitecto observa en la figura 4-35 que 2 procesos

	MS1						
	Caso de Uso 1				Caso de Uso 2		
	PF1	PF2	PF3	PF4	PF5	PF6	PF7
Volatilidad	No Volatil	No Volatil	No Volatil	No Volatil	No Volatil	No Volatil	No Volatil

	MS2						
	Caso de Uso 3				Caso de Uso 4		
	PF8	PF9	PF10	PF11	PF12	PF13	PF14
Volatilidad	Volatil	No Volatil	Volatil	No Volatil	No Volatil	No Volatil	No Volatil

Figura 4-35: Clasificando procesos funcionales conforme a su volatilidad.

funcionales del Caso de Uso 3 se clasifican como volátiles. Por ésto el decide crear un nuevo microservicio con los procesos funcionales del Caso de Uso 3. La salida de esta actividad se puede observar en la figura 4-36.

MS1						
Caso de Uso 1				Caso de Uso 2		
PF1	PF2	PF3	PF4	PF5	PF6	PF7

MS2			
Caso de Uso 3			
PF8	PF9	PF10	PF11

MS3		
Caso de Uso 4		
PF12	PF13	PF14

Figura 4-36: Salida de la actividad “Separar Microservicios: Diferencia de Volatilidad” para ejemplo de la figura 4-35.

4.6. Resolver contradicciones y Persistencia

Este es el último subproceso del método DISC. Este recibe las salidas de los subprocesos anteriores **Analizar perspectiva de Infraestructura**, **Analizar perspectiva de Seguridad** y **Analizar perspectiva de Calidad**. El subproceso ayuda a la toma de decisiones para 2 situaciones específicas.

La primera situación es que al recibir las 3 salidas de los subprocesos pasados se pueden encontrar contradicciones entre ellas. Se puede dar el caso donde, al terminar el subproceso **Analizar perspectiva de Seguridad**, los procesos funcionales del Caso de Uso 1 y del Caso de Uso 2 pertenecen a un mismo microservicio (ver figura 4-37). También se puede dar el caso donde, al terminar el subproceso **Analizar pers-**

pectiva de Calidad, los procesos funcionales del Caso de Uso 1 y del Caso de Uso 2 pertenezcan a microservicios diferentes (ver figura 4-38). El arquitecto de software debe decidir cual de estas dos salidas es la más adecuada. Este subproceso ayuda al arquitecto a tomar este tipo de decisiones.

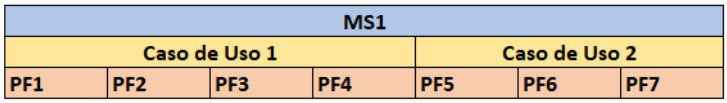

Figura 4-37: Salida del subproceso Analizar perspectiva de Seguridad.

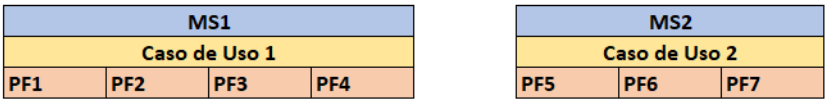

Figura 4-38: Salida del subproceso Analizar perspectiva de Calidad.

La otra situación de la que se encarga este subproceso es de definir qué microservicios van a ser los encargados de manejar la persistencia de qué grupos de datos. En la sección 2.1 de esta tesis se menciona que cada microservicio tiene su propia base de datos.

Suponiendo que dos microservicios MS1 y MS2 necesitan escribir un grupo de datos GD1, se debe de elegir cuál de estos dos microservicios va a manejar la persistencia de datos de GD1. Si se decide que MS1 maneja la persistencia de datos de GD1, entonces cuando MS2 necesita escribir GD1, éste debe comunicarse con MS1 para realizar su escritura. De igual manera, si se decide que MS2 maneja la persistencia de GD1, entonces MS1 tiene que comunicarse con MS2 cuando desea escribir GD1.

Para resolver estas situaciones se deben realizar 2 actividades. Es decisión del arquitecto de software si las actividades se realizan secuencialmente o en paralelo. Las actividades de este subproceso se pueden observar en la figura 4-39.

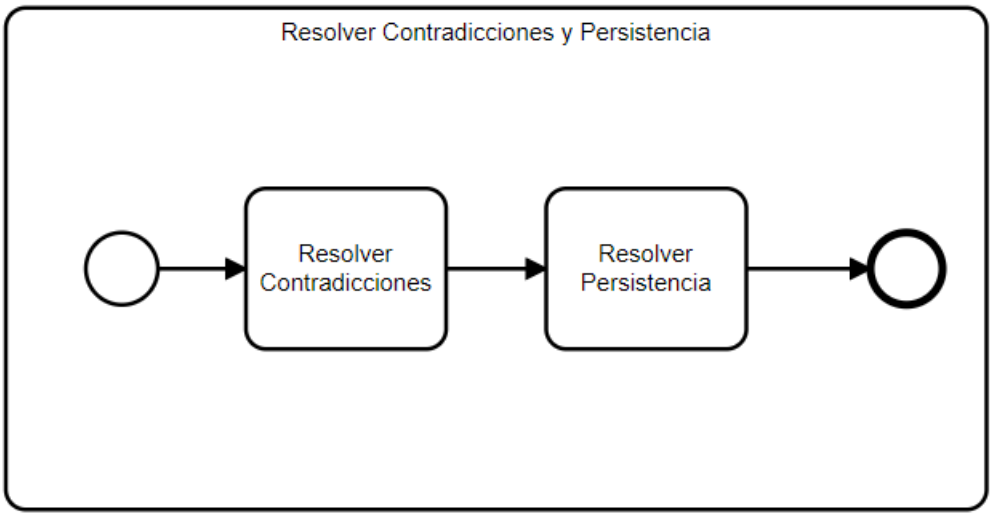

Figura 4-39: Subproceso “Resolver contradicciones y Persistencia” expandido.

4.6.1. Resolver Contradicciones

Para este método, una contradiccion se refiere al caso donde 2 o más salidas separaron los microservicios de manera que es imposible mantener ambas salidas. Este caso se observa en las figuras 4-37 y 4-38. En este ejemplo, se debe tomar la decisión de si se mantienen los microservicios de la figura 4-37 o los de la figura 4-38.

Para tomar estas decisiones, se debe tomar en cuenta la **razón por la que los procesos funcionales se separaron o se mantuvieron juntos.** Si se obtiene una contradicción es porque durante el subproceso “Analizar perspectiva de dominio” ciertos procesos funcionales se fusionaron en un mismo microservicio. Posteriormente estos mismos procesos funcionales se separaron en un cierto subproceso; y no se separaron en otro cierto subproceso. El arquitecto de software debe analizar las ventajas y desventajas de separarlos o de mantenerlos unidos conforme al subproceso que los separa y al que los mantiene unidos.

A diferencia de las actividades de los otros subprocesos, esta actividad no tiene pasos fijos a seguir porque las decisiones se toman dependiendo de las prioridades del sistema y el juicio del arquitecto de software. La salida que se muesta en la figura 4-37 es adecuada desde la perspectiva de seguridad. Sin embargo, desde la perspectiva de calidad, la salida que se muestra en la figura 4-38 es adecuada. Al final es decisión del

arquitecto de software qué salida es la más adecuada para el contexto del sistema.

4.6.2. Resolver Persistencia

El problema que esta actividad quiere resolver es el decidir qué grupos de datos va a persistir cada microservicio. Un microservicio tiene su propia base de datos. Si el microservicio necesita datos que no están en su base de datos, se debe comunicar con otro microservicio para obtener esos datos.

Durante esta actividad se decide qué grupos de datos van a ser persistentes por un microservicio y qué grupos de datos se tienen que obtener o escribir mediante la comunicación con otro microservicio.

Se presenta una lista de algunos factores, mas no todos, que se deben tomar en cuenta para la toma de estas decisiones:

1. **Latencia.** En el la sección 2.1.3 de esta tesis se menciona un criterio de acoplamiento llamado **CC-5 Latencia** en el que se menciona que, si hay nanoentidades con requerimientos de alto desempeño, estas nanoentidades deben pertenecer al mismo microservicio con el fin de evitar llamadas remotas y, con esto, evitar latencias. Por lo que si hay un proceso funcional con requerimientos de alto desempeño, desde el punto de vista de latencia, se le debe dar prioridad al microservicio al que pertenece este proceso funcional para que persista los grupos de datos que el proceso funcional lee y/o escribe.

2. **Criticalidad de Consistencia.** En la sección 2.1.3 de esta tesis se menciona un criterio de acoplamiento llamado **CC-6 Criticalidad de consistencia** donde se menciona que algunos datos pierden su valor cuando tienen inconsistencias mientras que otros datos son más tolerantes a inconsistencias. Si un proceso funcional escribe uno o más grupos de datos de consistencia crítica, desde el punto de vista de la consistencia de los datos, se le debe dar prioridad al microservicio al que pertenece este proceso funcional para que persista los grupos de datos que este proceso funcional escribe.

3. **Volatilidad Estructural.** Se menciona en la sección 4.5.1 que la descomposición de servicios debe ser realizada de una manera que favorezca el compartir las nanoentidades no-estructuralmente volátiles. Mientras más grupos de datos persiste un microservicio, hay mayor probabilidad de que otros microservicios se tengan que comunicar con este para leer o escribir datos. Por esto, si se debe de decidir entre que un microservicio no-estructuralmente volátil persista un cierto grupos de datos o que lo haga un microservicio estructuralmente volátil, es aconsejable que el microservicio no-estructuralmente volátil maneje la persistencia de ese grupo de datos.

4. **Relación de escritura entre proceso funcional y grupos de datos.** Entre menos comunicación de escritura de datos haya entre microservicios es mejor. Esto porque se pueden disminuir los problemas de consistencia de datos. Por lo que si hay 2 procesos funcionales en microservicios diferentes, uno de ellos escribe un grupo de datos y el otro lee ese mismo grupo de datos, se debe dar prioridad al que escribe para persistir ese grupo de datos.

5. **Relación de lectura entre proceso funcional y grupos de datos.** Una relación de lectura tiene menor prioridad que una relación de escritura al momento de resolver persistencia. Sin embargo, puede haber situaciones donde la relación de lectura se tome en cuenta al momento de dar prioridad de persistencia de un cierto grupo de datos a un microservicio.

6. **Requerimientos de Seguridad.** Si uno o varios grupos de datos tienen requerimientos fuertes de seguridad, entonces los procesos funcionales que acceden a los datos directamente deberían ser igual de seguros, con el fin de evitar robo de datos. Tomando esto en cuenta, tiene sentido mantener estos datos y procesos funcionales en un mismo microservicio, para que un equipo con experiencia en seguridad se haga cargo del desarrollo y mantenimiento.

Se puede dar el caso donde existan diferentes razones para que uno u otro microservicio maneje la persistencia de datos. La figura 4-40 muestra un ejemplo de

esto. En este ejemplo existe un microservicio MS1 que tiene procesos funcionales con requerimiento de alto desempeño y lee GD1, y existe otro microservicio que tiene una relación de escritura con GD1. Ambos microservicios cuentan con razones para persistir GD1; MS1 por latencia y MS2 por la relación de escritura. En estos casos se pueden hacer 2 cosas, es decisión del arquitecto de software el qué hacer.

1. **Priorizar un microservicio.** La primera opción es darle prioridad a un microservicio. Para el ejemplo de la figura 4-40 probablemente la latencia sea más importante que la relación de escritura, por lo que el arquitecto de software podría decidir que MS1 persista a GD1.

2. **Fusionar los microservicio.** La segunda opción es fusionar los dos microservicios. Esta decisión disminuiría el número de microservicios y aumentaría la granularidad del microservicio que se fusiona.

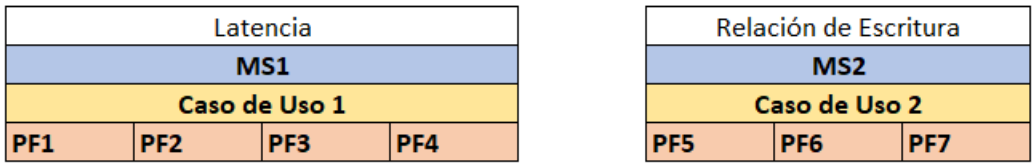

Figura 4-40: Ejemplo: MS1 lee un grupo de datos GD1 y tiene requerimiento no funcional de alto desempeño; Los procesos funcionales de MS2 escriben el grupo de datos GD1.

4.6.3. Ejemplo

Se continúa con el ejemplo de las secciones anteriores. Por practicidad, se vuelven a presentar las salidas de los subprocesos **Analizar perspectiva de Infraestructura**, **Analizar perspectiva de Seguridad** y **Analizar perspectiva de Calidad** en las figuras 4-41, 4-42 y 4-43. Para ayudar a la toma de decisiones se recuerda qué decisiones ha tomado el arquitecto y el porque se tomaron esas decisiones.

Durante el subproceso de **Analizar Perspectiva de Infraestructura** se han tomado las siguientes decisiones:

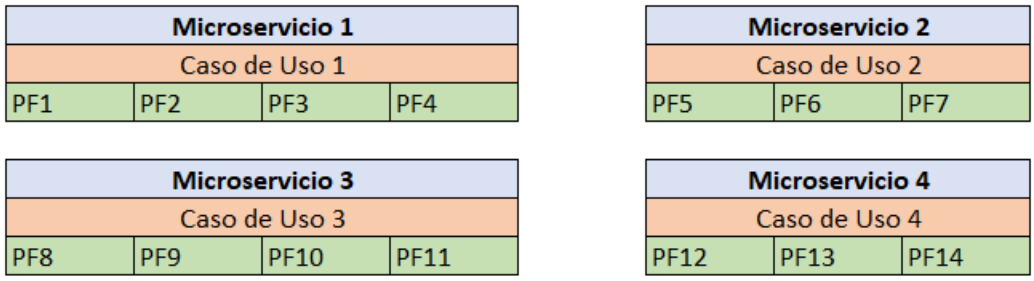

Figura 4-41: Salida del subproceso **Analizar perspectiva de Infraestructura**.

- **Nivel de Almacenamiento** MS1 y MS2 se separan por tener un nivel de almacenamiento diferente.
- **Tráfico diferente en la red** MS3 y MS4 se separan por tener un tráfico diferente en la red. MS1 y MS2 tiene un tráfico similar en la red.

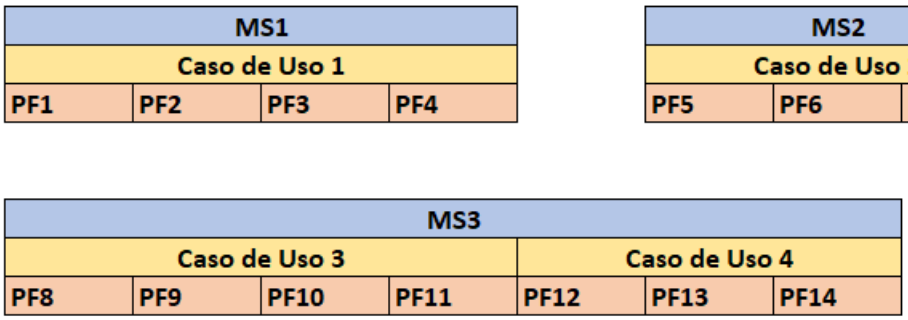

Figura 4-42: Salida del subproceso **Analizar perspectiva de Seguridad**.

Durante el subproceso de **Analizar Perspectiva de Seguridad** se han tomado las siguientes decisiones:

- **Criticalidad de Seguridad**: MS1 y MS2 se separan por tener una criticalidad de seguridad ligeramente diferente.
- **Contextualidad de Seguridad**: No hay cambios.

Durante el subproceso de **Analizar Perspectiva de Calidad** se han tomado las siguientes decisiones:

- **Diferencia de Volatilidad**: MS2 y MS3 se separan porque MS2 esta clasificado como volátil, y MS3 no.

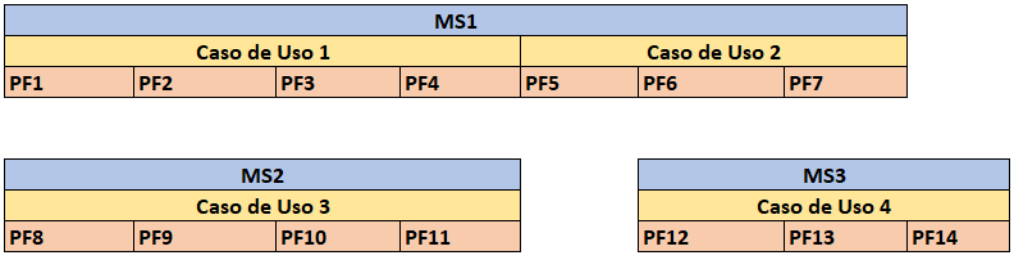

Figura 4-43: Salida del subproceso **Analizar perspectiva de Calidad**.

El arquitecto observa algunas diferencias entre las salidas de los 3 subprocesos, sin embargo, tambíen observa similitudes. Los procesos funcionales del Caso de Uso 1 y el del Caso de Uso 2 han separado por nivel de almacenamiento y por criticalidad de seguridad. El arquitecto observa también considera que los procesos funcionales del Caso de Uso 3 y del Caso de Uso 4 se separaron tanto por tráfico diferente en la red como por volatilidad estructural. Observando esto, el arquitecto decide mantener separados los procesos funcionales del caso de uso 1, del caso de uso 2, del caso de uso 3 y del caso de uso 4.

Posteriormente, se pasa a resolver el problema de la persistencia. Para esto se necesita conocer la relación de lectura y escritura entre los procesos funcionales y los grupos de datos. Estas relaciones se pueden observar en la figura 4-5 y, por practicidad, en la figura 4-44. Para este ejemplo se asume que ni un proceso funcional tiene requerimientos de alto desempeño ni de criticalidad de consistencia, por lo que solo se toman en cuenta los factores volatilidad estructural y de relaciones de escritura entre proceso funcional y grupos de datos.

Conociendo la informacion de la figura 4-44 y la volatilidad de los microservicios el arquitecto genera una lista de razones positivas y negativas para que un microservicio maneje o no la persistencia de un grupo de datos. La lista de estas razones se puede observar en la figura 4-45. En esta figura, donde dice $+GDx$ se explica una razón a favor de que ese microservicio persista el grupo de datos x. Al contrario, donde dice $-GDx$ se explica una razón en contra de que ese microservicio persista el grupo de datos x.

Con esta información el arquitecto toma las siguientes decisiones:

Procesos Funcionales/Grupos de datos	GD1	GD2	GD3	GD4
PF1	W		R	
PF2	R			
PF3	R			R
PF4	W			
PF5	W	W		R
PF6		R		
PF7		R		
PF8	R		W	
PF9		R	W	
PF10			R	
PF11			R	
PF12				R
PF13			R	W
PF14				R

Figura 4-44: Relaciones de lectura y escritura entre grupos de datos y procesos funcionales.

MS1	
Grupo de Datos	**Razones**
+GD1	Tanto PF1 y PF4 escriben GD1

MS2	
Grupo de Datos	**Razones**
+GD1	PF5 escribe GD1
+GD2	PF5 escribe GD2

MS3	
Grupo de Datos	**Razones**
+GD3	Tanto PF8 y PF9 escriben GD3
-GD3	Volatilidad

MS3	
Grupo de Datos	**Razones**
+GD4	PF13 escribe GD4

Figura 4-45: Razones a favor y en contra de que ciertos microservicios persistan ciertos grupos de datos.

- **GD1.** Tanto MS1 como MS2 tienen razones a favor para persistir este grupo de datos. La diferencia es que en MS1 dos procesos funcionales escriben GD1 mientras que en MS2 solo uno escribe. Se puede decidir fusionar MS1 y MS2 (se tendría que reflexionar en cómo afecta esto a las decisiones tomadas en la actividad de resolver contradicciones) pero para este ejemplo se decide mantener MS1 separado de MS2 y que MS1 maneje la persistencia de GD1.

- **GD2.** Solo MS2 tiene razones a favor para persistir GD2, por lo MS2 maneja la persistencia de GD2.

- **GD3.** MS3 tiene razones a favor y en contra para persistir GD3. El arquitecto debe decidir sobre permitir que GD3 sea persistido por otro microservicio (tal vez uno que lea GD3) o permitir que el microservicio volátil maneje la persistencia. Para este caso se permite que MS3 maneje la persistencia de GD3.

- **GD4.** Solo MS4 tiene razones a favor para persistir GD4, por lo MS4 maneja la persistencia de GD4.

MS1		
Procesos Funcionales	**Persiste**	**Se comunica**
PF1	GD1	MS3
PF2		
PF3		MS4
PF4	GD1	

MS2		
Procesos Funcionales	**Persiste**	**Se comunica**
PF5	GD2	MS1, MS4
PF6		
PF7		

MS3		
Procesos Funcionales	**Persiste**	**Se comunica**
PF8	GD3	MS1
PF9	GD3	MS2
PF10		
PF11		

MS4		
Procesos Funcionales	**Persiste**	**Se comunica**
PF12		
PF13	GD4	MS3
PF14		

Figura 4-46: Salida del método DISC.

Al finalizar, se obtiene la figura 4-46 donde se puede observar el resultado de separar el sistema en microservicios mediante el método DISC. Se puede ver qué procesos funcionales pertenecen a cada microservicio, cuáles son los procesos funcionales que persisten datos y cuáles procesos funcionales se comunican con otros microservicios para completar su funcionalidad.

4.7. Métricas para Microservicios

Se proponen 2 nuevas métricas para microservicios. La primera de ellas se utiliza para medir la granularidad de un microservicio, y la otra métrica se utiliza para medir acoplamiento entre microservicios.

4.7.1. Métrica de Granularidad

En el artículo de Shadija [19] se da una definición de granularidad de microservicios. Esta definición menciona que la granularidad es lo mismo que el tamaño funcional. Sin embargo, en ese mismo artículo se menciona que el tamaño de la funcionalidad es equivalente a la complejidad del código o al número de casos de uso implementados en un microservicio.

Los problemas que se encuentran con la definición de Shadija[19] son los siguientes:

1. Se menciona que la granularidad es lo mismo que el tamaño funcional, sin embargo, no se utilizan métodos estandarizados ya existentes para medir el tamaño funcional del software.

2. Es falso que el tamaño funcional del software sea equivalente a la complejidad del código o al número de casos de uso de un software [6].

El concepto de granularidad de microservicios se menciona múltiples veces en la literatura. Incluso, se menciona que es un problema el no saber cuál es la granularidad "correcta" de un microservicio.

Para responder la pregunta ¿Qué tan micro debe ser un microservicio? Primero se tiene que responder a la pregunta ¿Cómo se calcula el tamaño de un microservicio? Si se utiliza la definición de granularidad de Shadija[19], el resultado de la medición va a depender de la persona que realiza la medición. Por esto se afirma que la definición de granularidad que presenta Shadija es subjetiva. Por lo tanto, la respuesta a las preguntas "¿Qué tan micro debe ser un microservicio?" y "¿Cuál es la granularidad de un microservicio?" pasaría a ser "Depende de la persona que mide el software".

La métrica propuesta es utilizar el concepto de "Proceso Funcional" del estándar COSMIC, presentado en la sección 2.4 de esta tesis, para resolver esta problemática. Se propone que la granularidad de un microservicio sea igual a el número de procesos funcionales que pertenecen a ese microservicio. De tal manera que, solo con contar el número de procesos funcionales, se puede saber la granularidad de un microservicio.

4.7.2. Métrica de Acoplamiento

En el paradigma Orientado a Objetos se busca que el software presente bajo acoplamiento y alta cohesión, ya que son caracteristicas de Software de Calidad. Acoplamiento se refiere a la interdependencia que existe entre diferentes objetos, y cohesión es el grado de consistencia conceptual dentro de un mismo objeto. De tal manera que un Software con bajo acoplamiento es un software donde cada uno de sus objetos depende poco o nada de otros objetos [21].

La misma idea de bajo acoplamiento hace sentido para MSA. Una alta interdependencia entre microservicios puede causar: Latencia y tráfico en la red, alta interdependencia entre equipos de desarrollo, entre otros problemas.

Esta tesis presenta una propuesta para medir el acoplamiento entre microservicios utilizando conceptos del método de medición COSMIC. La idea base es que, cuando un microservicio MS1 hace una petición HTTP (o mediante otro protocolo) a otro microservicio MS2, esta petición se realiza porque MS1 quiere mandar (salida) o recibir (entrada) datos de MS2.

Para el caso de esta métrica, cuando se dice que MS1 está acoplado a MS2 se quiere dar a entender que MS1 necesita a MS2 para completar una cierta porción de su funcionalidad. Sin embargo, esto no necesariamente quiere decir que MS2 esté acoplado a MS1, porque probablemente MS2 no necesite a MS1 para realizar su funcionalidad.

Se da un ejemplo para que quede claro: Se supone que MS1 es un microservicio que se encarga de las ventas en un supermercado, y MS2 es un microservicio que su única funcionalidad es dar la hora a cualquier microservicio que lo requiera. Al momento de que MS1 quiere dar de alta una venta necesita guardar en su base de datos la hora de la venta. Por lo tanto MS1 se comunica con MS2 para obtener la hora actual. Para poder finalizar el alta de la venta MS1 necesita a MS2, sin embargo, MS2 no requiere de MS1 para realizar su funcionalidad (dar la hora). En este caso se diría que MS1 está acoplado a MS2, sin embargo MS2 no está acoplado a MS1.

Se decide utilizar el concepto de acoplamiento de esta manera porque, normalmen-

te, los microservicios ofrecen sus servicios mediante una API HTTP [12]. Estas APIs permiten que el software pueda ofrecer sus servicios a múltiples clientes. Siguiendo con el ejemplo anterior, MS1 es cliente de MS2, sin embargo, MS2 puede tener 1000 clientes más. Se considera que no tiene sentido el pensar que MS2 está acoplado a 1000 clientes por el simple hecho de que los 1000 clientes utilizan sus servicios, más aún si desde la perspectiva de MS2 no interesa quien es el que está utilizando el servicio.

El manual de medición de COSMIC [5] explica el cómo medir software, mediante el conteo de los movimientos de datos que se realizan en los diferentes procesos funcionales. Existen ciertas reglas sobre cúando sí y cúando no un cierto movimiento de datos se toma en cuenta para la medición. La explicación de estas reglas están fuera del contexto de la tesis. La métrica de acoplamiento que propone la tesis se basa en tomar esas mismas reglas de COSMIC y aplicarlas al contexto de la tesis. De manera que la medición de acoplamiento de un microservicio MS1 a un microservicio MS2 se calcula tomando en cuenta las entradas y salidas que se realizan en los diferentes procesos funcionales de MS1, y que van de MS1 a MS2.

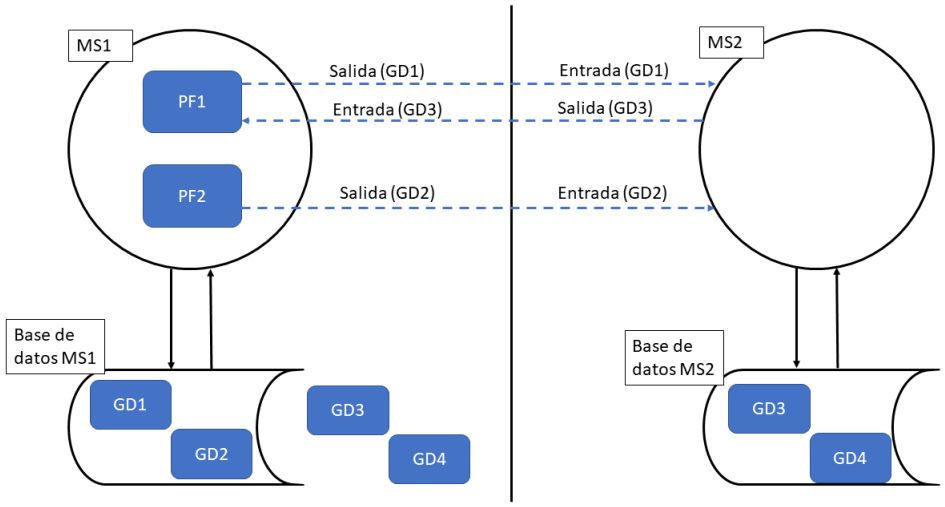

Figura 4-47: Comunicación entre MS1 y MS2. Las lineas punteadas representan llamadas HTTP, donde se escriben o leen ciertos grupos de datos.

Por ejemplo, en la figura 4-47 PF1 necesita realizar un movimiento de datos de salida a MS2 y un movimiento de datos de entrada de MS2 para completar su funcio-

nalidad. Es decir, 2 movimientos de datos en total en PF1 de MS1 a MS2. También se puede observar que PF2 necesita un movimiento de datos de salida a MS2 para completar su funcionalidad. Es, 1 movimiento de datos en total en PF2 de MS1 a MS2. Sumando los movimientos de datos de los dos procesos funcionales, se puede decir que el resultado de la medición de acoplamiento de MS1 a MS2 es de 3.

Medir los microservicios de esta manera permite obtener un valor objetivo de la dependencia que tiene un cierto microservicio hacia otro microservicios del sistema. Entre menor sea la dependencia más bajo el acoplamiento entre los microservicios.

4.8. Resumen

El método DISC se compone de 5 etapas: Analizar perspectiva de Dominio, Analizar perspectiva de Infraestructura, Analizar perspectiva de Seguridad, Analizar perspectiva de Calidad, y Resolver Contradicciones. Este método presenta un proceso ordenado para separar un sistema en microservicios, toma en cuenta los requerimientos no funcionales del sistema, y presenta el enfoque de medir la granularidad de microservicio utilizando conceptos pertenecientes al estandar COSMIC de medición de software.

En este capítulo se explica a detalle cada una de las etapas del método DISC, así como un ejemplo, con el fin de facilitar la comprensión del método. La explicación de las etapas avanza simultáneamente con el ejemplo. Al final se presentan dos propuestas, basadas en conceptos del método de medición COSMIC, para medir granularidad y acoplamiento de microservicios.

Capítulo 5

Caso de Estudio

En este capítulo se desarollan 2 casos de estudio utilizando el método DISC. Los casos de uso son llamados “Cargo Tracker” y “Trading System”. Se comparan las salidas del método DISC con las salidas del método Service Cutter desarrollado por Gysel [14]. Para este caso de estudio, los datos de entrada para el método Service Cutter y para el método DISC son los mismos. Por esto se considera válido realizar la comparación entre los dos métodos.

5.1. Caso de Estudio: Cargo Tracker

En el libro de Domain-Driven Design de Eric Evans [8] se presenta un ejemplo ficticio del módelo de un sistema. En este ejemplo una compañía de envíos de carga (Ver figura 5-1) está desarrollando un software que, al principio del proyecto ficticio, cuenta con tres funcionalidades básicas.

1. Rastrear la carga de un cliente.
2. Reservar una carga por adelantado.
3. Enviar facturas a los clientes de forma automática cuando la carga llegue a algún punto en su manejo.

Mientras se avanza en el libro se añaden, explícita e implícitamente, nuevas funcionalidades. Esto representa un problema ya que no hay un documento de reque-

Figura 5-1: Tipo de cargas para las que se desarrolla el sistema.

rimientos que permita conocer todas las funcionalidades del software. Utilizando la información presente en el libro, se ha realizado una implementación del sistema expuesto por Evans[1]. En el artículo de Gysel [14] se realiza ingeniería inversa a esta implementación para obtener los requerimientos.

La ingeniería inversa da un resultado de 9 Casos de Uso. En el libro [8] se hace mención, ya sea explícita o implícitamente, a las funcionalidades de los 9 Casos de Uso. No es claro cuando añade un nuevo requerimiento al sistema. Por esto no se puede saber a ciencia cierta si la implementación a la que se realiza ingeniería inversa tiene toda la funcionalidad que Evans quería expresar.

Los grupos de datos encontrados durante la ingeniería inversa son:

- *Carga.* Cada una de las "cajas" observadas en la figura 5-1 representa una carga [14, 8].
- *Especificación de la ruta.* Contiene la información de origen y destino de una carga [14, 8].
- *Locación.* Representa un lugar.
- *Etapa.* (traducción de leg) No siempre hay rutas directas desde el origen de una carga hasta su destino. Por esto, las cargas tienen que llegar a otras locaciones

[1] https://sourceforge.net/projects/dddsample/

con el fin de transbordar y así acercarse al destino. El grupo de datos de etapa ofrece la información sobre a donde se puede llegar de manera directa desde una locación especifica. De manera que, para generar una ruta, se necesitan una cadena de etapas. La cadena de etapas permite conocer todos los transbordes que hace una carga para llegar a su destino [14, 8].

- *Viaje.* Un viaje representa el concepto de mover una o más cargas de un origen a un destino. Puede decirse que es la acción de utilizar una ruta [14, 8].

- *Movimiento del transportador.* Representa un movimiento directo (sin transbordar) de una locación a otra. Durante un viaje se pueden realizar varios movimientos de portador [14, 8].

- *Itinerario.* Representa un itinerario [14, 8].

- *Entrega.* Incluye el estado del transporte, tiempo estimado de llegada, estado de la ruta, etc. [14, 8]

- *Evento de manejo.* Representa una acción en el mundo real sobre algo que se hizo con la carga (cargar o descargar por ejemplo) [14, 8].

Los casos de usos encontrados son:

1. **Ver seguimiento de la carga.** Durante este caso de uso, el sistema lee los grupos de datos de *carga*, *evento de manejo*, *entrega*, *viaje* y *especificación de la ruta*. No se realizan movimientos de datos de escritura [14, 8].

2. **Ver cargas.** En este caso de uso se leen los grupos de datos de *carga*, *especificación de la ruta*, *entrega itinerario*. No se realizan movimientos de datos de escritura [14, 8].

3. **Reservar una carga.** Para este caso de uso se lee el grupo de datos de *locación*. Se escriben los grupos de datos de *carga* y *especificación de la ruta* [14, 8].

4. **Cambiar el destino de la carga.** En este caso de uso se leen los grupos de datos de *carga*, *especificación de la ruta*. Ademas se escribe el grupo de datos de *especificación de la ruta* [14, 8].

5. **Enrutar una carga.** Durante este caso de uso se leen los grupos de datos de *carga*, *especificación de la ruta*, *locación*, *viaje* y *movimiento del portador*. También se escriben los grupos de datos de *itinerario* y *etapa* [14, 8].

6. **Crear una locación.** Para este caso de uso se escriben los grupos de datos de *locación* [14, 8].

7. **Crear un viaje (ruta).** En este caso de uso solo se escribe el grupo de datos de *viaje* [14, 8].

8. **Agregar un movimiento del portador.** Para este caso de uso se lee el grupo de datos de *viaje* y se escribe el grupos de datos de *movimiento del portador* [14, 8].

9. **Administrar evento de carga.** Durante este caso de uso se leen los grupos de datos de *viaje* y *carga*. Se escriben los grupos de datos de *eventos de manejo* y *entrega* [14, 8].

5.1.1. Analizar Perspectiva de Dominio

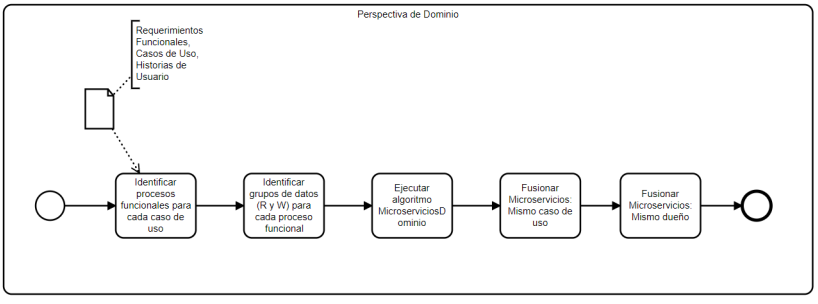

Figura 5-2: Subproceso “Analizar perspectiva de Dominio” expandido.

Identificar procesos funcionales

La información que se tiene no es suficiente para obtener los procesos funcionales de manera correcta. Para cuestiones de esta tesis se toma en cuenta cada caso de uso como un proceso funcional. Lo más adecuado sería tener documentos de requerimientos de software para obtener de manera correcta los procesos funcionales, pero el libro de Evans no incluye nada parecido.

Identificar relaciones de lectura y escritura

La información sobre las lecturas y escrituras del sistema se obtienen del artículo de Gysel. Se puede observar esta información en forma de tabla en la figura 5-3. En la figura se observa algo extraño: que el grupo de datos *Etapa* es escrito pero nunca leído. Se desconce la razón de esto, para fines de este caso de estudio se maneja de esa manera.

Proceso Funcional/Grupo de Datos	GD1 - Carga	GD2 - Especificacion de la ruta	GD3 - Locación	GD4 - Etapa	GD5 - Viaje	GD6 - Movimiento del Transportador	GD7 - Itinerario	GD8 - Entrega	GD9 - Evento de Manejo
PF1 - Ver seguimiento de la carga	R	R			R			R	R
PF2 - Ver cargas	R	R					R	R	
PF3 - Reservar una carga	W	W	R						
PF4 - Cambiar el destino de la carga	R	W,R							
PF5 - Enrutar una carga	R	R	R	W	R	R	W		
PF6 - Crear una locación			W						
PF7 - Crear un viaje					W				
PF8 - Agregar movimiento del portador			R			W			
PF9 - Administrar evento de carga	R		R					W	W

Figura 5-3: Relaciones entre procesos funcionales y grupos de datos. W para relación de escritura y R para relación de lectura.

Ejecutar algoritmo MicroserviciosDominio

El algoritmo MicroserviciosDominio se encuentra en la sección 4.2.3 de la tesis. Al realizar este algoritmo se obtienen los primeros microservicios del método DISC. Estos microservicios van cambiando mientras avanza el método. Los microservicios obtenidos se pueden observar en la figura 5-4.

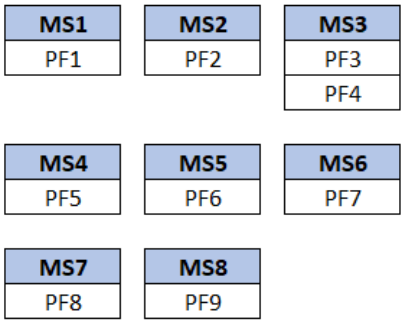

Figura 5-4: Salida de la subtarea “Ejecutar algoritmo MicroserviciosDominio”.

Fusionar Microservicios: Mismo caso de uso

Esta sección se omite porque se toma en cuenta cada caso de uso como un proceso funcional.

Fusionar Microservicios: Mismo dueño

Se conoce la información presente en la figura 5-5. En esta figura se pueden observar los dueños de cada proceso funcional. Durante este paso se fusionan los microservicios que tienen procesos funcionales que pertenecen a un mismo dueño. Al realizar esta tarea, el resultado es el que se muestra en la figura 5-6.

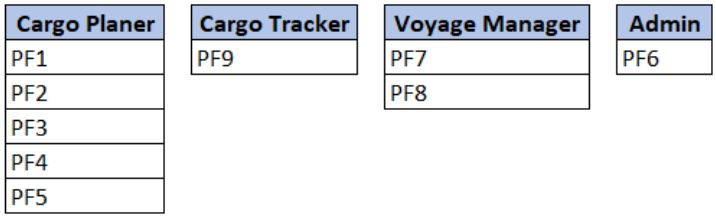

Figura 5-5: Información sobre qué procesos funcionales pertenecen a qué dueños.

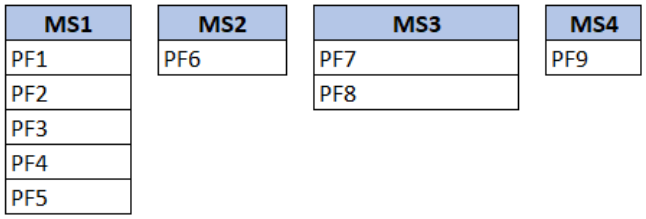

Figura 5-6: Salida de la subtarea “Fusionar Microservicios: Mismo dueño”.

5.1.2. Analizar Perspectiva de Infraestructura

Para este caso de estudio no existe la información necesaria para realizar este subproceso del método.

5.1.3. Analizar Perspectiva de Seguridad

Para este caso de estudio no existe la información necesaria para realizar este subproceso del método.

5.1.4. Analizar Perspectiva de Calidad

Separar Microservicios: Diferencia de Volatilidad Estructural

Para esta tarea se conoce la información presente en la figura 5-7. Sin embargo, el proceso funcional PF6 no comparte microservicio con otros procesos funcionales (ver figura 5-6) por lo que no se generan cambios.

	Volatilidad Estructural
PF6 - Crear una locación	Casi nunca

Figura 5-7: Información conocida sobre volatilidad estructural.

5.1.5. Resolver decisiones contradictorias

Resolver Contradicciones

Para este caso de estudio no hay diferencias entre las salidas de los tres subprocesos anteriores (**Analizar perspectiva de Infraestructura**, **Analizar perspectiva de Seguridad** y **Analizar perspectiva de Calidad**). Por esto mismo no hay contradicciones que resolver.

Resolver Persistencia

Como se menciona en la sección 4.6.2 de esta tesis, para tomar estas decisiones se deben tomar en cuenta los siguientes factores: Latencia, Criticalidad de consistencia,

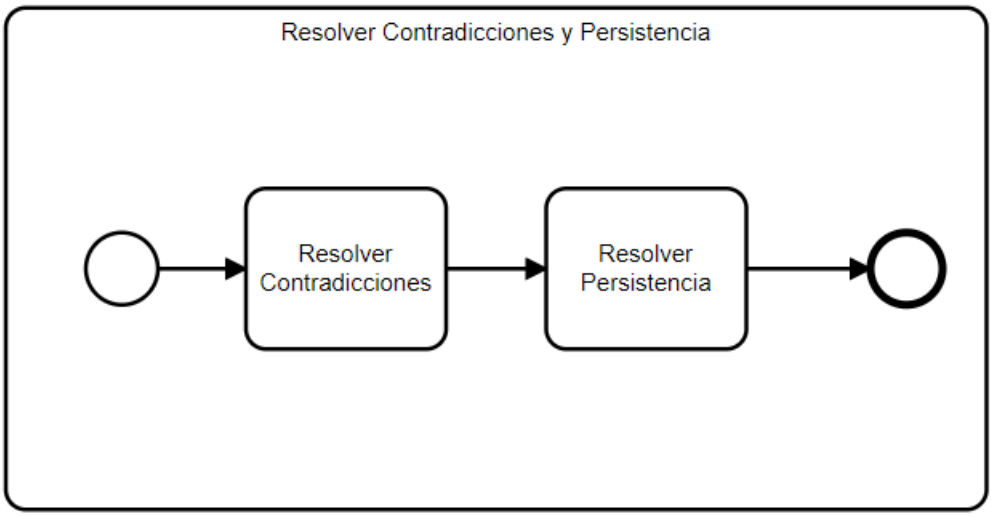

Figura 5-8: Subproceso “Resolver contradicciones y Persistencia” expandido.

volatilidad estructural y relaciónes de lectura y escritura entre proceso funcional y grupos de datos.

Se puede dar el caso donde existan diferentes razones para que uno u otro microservicio maneje la persistencia de datos de un grupo de datos. En estos casos se pueden hacer 2 cosas: Priorizar un microservicio o fusionar los microservicios.

Proceso Funcional/Grupo de Datos	GD1 - Carga	GD2 - Especificacion de la ruta	GD3 - Locación	GD4 - Etapa	GD5 - Viaje	GD6 - Movimiento del Transportador	GD7 - Itinerario	GD8 - Entrega	GD9 - Evento de Manejo
PF1 - Ver seguimiento de la carga	R	R			R			R	R
PF2 - Ver cargas	R	R					R	R	
PF3 - Reservar una carga	W	W	R						
PF4 - Cambiar el destino de la carga	R	W,R							
PF5 - Enrutar una carga	R	R	R	W	R	R	W		
PF6 - Crear una locación			W						
PF7 - Crear un viaje					W				
PF8 - Agregar movimiento del portador			R			W			
PF9 - Administrar evento de carga	R		R					W	W

Figura 5-9: Relaciones entre procesos funcionales y grupos de datos. W para relación de escritura y R para relación de lectura.

Conociendo la información de la figura 5-9 se puede generar una lista de razones positivas y negativas para que un microservicio maneje o no la persistencia de un grupo de datos. Para este caso de estudio la lista de razones se puede observar en la figura 5-10. En esta figura, hay una columna llamada “Grupo de datos”, aquí se clasifican las situaciones a favor o en contra de que un cierto grupo de datos sea

persistido por un microservicio. En las fila donde dice *+GDx* se explica una razon a favor de que ese microservicio persista el grupo de datos *x*. Al contrario, la fila que dice *-GDx* se explica una razon en contra de que ese microservicio persista el grupo de datos *x*. Para este caso de estudio se maneja la clasificación *GDx*, esta representa razones ligeramente a favor de que el microservicio persista el grupo de datos *x* (principalemente que un microservicio lea el grupo de datos *x*).

MS1	
Grupo de Datos	**Razones**
+GD1	Es leido o escrito por todos los procesos funcionales
+GD2	Es leido o escrito por todos los procesos funcionales
GD3	Es leído por PF3 y PF5
+GD4	Es escrito por PF5
GD5	Es leido por PF1 y PF5
GD6	Es leido por PF5
+GD7	Es leido por PF2 y escrito por PF5
GD8	Es leido por PF1 y PF2
GD9	Es leido por PF1

MS2	
Grupo de Datos	**Razones**
+GD3	Es escrito por PF6

MS3	
Grupo de Datos	**Razones**
GD3	Es leido por PF8
+GD5	Es escrito por PF7
+GD6	Es escrito por PF8

MS4	
Grupo de Datos	**Razones**
GD1	Es leido por PF9
GD3	Es leido por PF9
+GD8	Es escrito por PF9
+GD9	Es escrito por PF9

Figura 5-10: Razones a favor y en contra de que ciertos microservicios persistan ciertos grupos de datos.

Con esta información se toman las siguientes decisiones:

- **GD1.** Solo MS1 tiene razones para persistir GD1. Por esto MS1 persiste GD1.
- **GD2.** Solo MS1 tiene razones para persistir GD2. Por esto MS1 persiste GD2.
- **GD3.** Solo MS2 tiene razones para persistir GD3. Por esto MS2 persiste GD3.
- **GD4.** Solo MS1 tiene razones para persistir GD4. Por esto MS1 persiste GD4.
- **GD5.** Solo MS3 tiene razones para persistir GD5. Por esto MS3 persiste GD5.
- **GD6.** Solo MS1 tiene razones para persistir GD6. Por esto MS3 persiste GD6.
- **GD7.** Solo MS1 tiene razones para persistir GD7. Por esto MS1 persiste GD7.
- **GD8.** Solo MS4 tiene razones para persistir GD8. Por esto MS4 persiste GD8.
- **GD9.** Solo MS4 tiene razones para persistir GD9. Por esto MS4 persiste GD9.

En la figura 5-10 se puede observar que MS1 lee datos de MS4, y que MS4 lee datos de MS1. Con esto, un arquitecto de software podría decidir fusionar los dos microservicios. Esta decisión queda en manos del arquitecto de software.

Al finalizar el método se obtiene la figura 5-11. Esta figura representa la salida de separar el sistema en microservicios mediante el método DISC. Se puede observar qué procesos funcionales pertenecen a cada microservicio, cuáles son los procesos funcionales que persisten datos y cuáles procesos funcionales se comunican con otros microservicios para realizar su funcionalidad.

MS1		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF1	MS3, MS4	GD5, GD8, GD9
PF2	MS4	GD8
PF3	MS2	GD3
PF4		
PF5	MS2, MS3	GD3, GD5, GD6

MS1
Persiste
GD1, GD2, GD4, GD7

MS2		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF6		

MS2
Persiste
GD3

MS3		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF7		
PF8	MS2	GD3

MS3
Persiste
GD5, GD6

MS4		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF9	MS1, MS2	GD1, GD3

MS4
Persiste
GD8, GD9

Figura 5-11: Salida del método DISC.

5.1.6. Comparación del método DISC y el método Service Cutter

El método Service Cutter propuesto por Gysel se explica en la sección 3.3 de esta tesis. El método utiliza algoritmos de agrupamiento para encontrar cortes de servicios. La implementación del método incluye 4 algoritmos de agrupamiento. Dos de estos algoritmos son no deterministas. Otro de estos algoritmos recibe como entrada

el número de microservicios deseados. El algoritmo restante es llamado Markov. El algoritmo de Markov es determinista y no recibe como entrada el número de microservicios deseados.

Comparar el método DISC utilizando otro algoritmo diferente al de Markov se vuelve complicado porque, gracias al no determinismo, se obtiene una salida diferente cada vez que se ejecuta el algoritmo, por lo que no se sabe con que salida realizar la comparación. Lo mismo para el algoritmo en el que se necesita como entrada el número de microservicios que tiene la salida, no se sabe con que número de microservicios realizar la comparación. Por esto, se decide utilizar el algoritmo de Markov para la comparación entre las salidas de Service Cutter y el método DISC.

Salida del método Service Cutter

La salida del método Service Cutter para el caso de estudio de Cargo Tracker es un grafo que se puede observar en la figura 5-12. Además del grafo, el método ofrece información de cada microservicio, un ejemplo de la información que se ofrece se puede observar en la figura 5-13. Con la información de salida se puede generar un formato simplificado con toda la información relevante, como se muestra en la figura 5-14.

Comparación de las salidas

Para este caso de estudio se pueden observar que las salidas son muy parecidas. Ambas salidas presentan 4 microservicios, ambas salidas tienen microservicios con granularidad parecida, y ambas salidas tienen microservicios con acoplamiento parecido (Ver sección 4.7 de esta tesis para información sobre cómo se está midiendo la granularidad y el acoplamiento). Incluso se puede observar que MS3 se encarga de los mismos procesos funcionales que el servicio B, lo mismo para MS2 y el servicio C.

La diferencia principal es que, en la salida del método DISC, el microservicio que tiene mayor granularidad y mayor acoplamiento es MS1 (Granularidad = 5 y acoplamiento = 8). En cambio, para el método Service Cutter, el microservicio que tiene mayor granularidad es el servicio D (Granularidad = 4), y el microservicio que

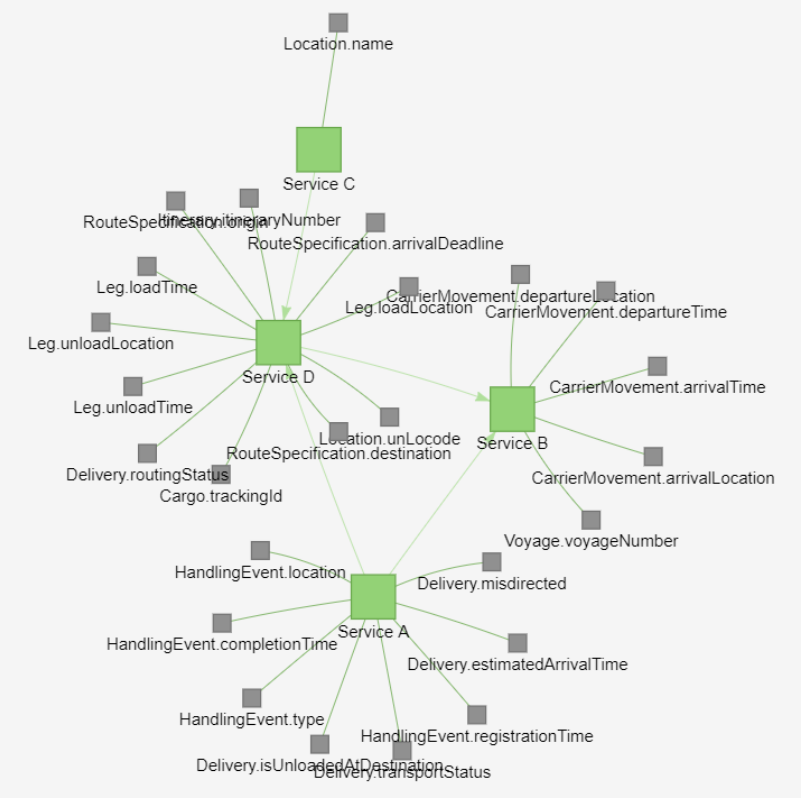

Figura 5-12: Salida del método Service Cutter.

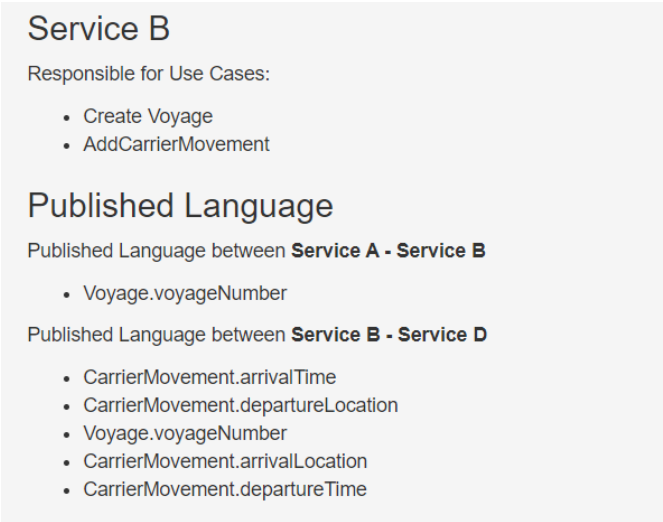

Figura 5-13: Información que ofrece el método Service Cutter para cada Microservicio.

tiene mayor acoplamiento es el servicio A (acoplamiento = 8).

La salida del método DISC presenta ventajas en cuanto a que el criterio de acoplamiento de mismo dueño no se rompe. Todos los procesos funcionales que pertenecen

Service A		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF1	Service B, Service D	GD1, GD2, GD5, GD8
PF9	Service B, Service D	GD1, GD2, GD5, GD8

Service A
Persiste
GD8, GD9

Service B		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF7		
PF8		

Service B
Persiste
GD5, GD6

Service C		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF6	Service D	GD3

Service C
Persiste
GD3

Service D		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF2		
PF3		
PF4		
PF5	Service B	GD5, GD6

Service D
Persiste
GD1, GD2, GD3, GD4, GD7, GD8

Figura 5-14: Salida del método Service Cutter con formato simplificado.

a un mismo dueño están juntos, y no hay, en un mismo microservicio, procesos funcionales que pertenezcan a diferentes dueños. En cambio, la salida de Service Cutter tiene a PF1 y PF9, que pertenecen a diferentes dueños, en un mismo microservicio.

La salida del método Service Cutter presenta la ventaja de que su método con mayor granularidad tiene un bajo acoplamiento. Esto puede llevar a que el mantenimiento de esta pieza de software sea más sencilla. También el microservicio con mayor acoplamiento tiene una baja granularidad, lo que puede permitir que se mantenga simple la pieza de software altamente acoplada.

Una situación que se da al utilizar Service Cutter, y que se puede observar tanto en la figura 5-12 como en la figura 5-14, es que se puede dar el caso que un mismo grupo de datos sea persistido por dos o más microservicios. Por ejemplo GD3 (Locación) es persistido por el servicio C y por el servicio D. Tambíen GD8 (Entrega) es persistido por el servicio A y el servicio D.

Para lograr esto, lo que Service Cutter hace es separar un grupo de datos en atributos de datos, y definir que diferentes microservicios pueden persistir diferentes atributos de datos del mismo grupo de datos. Por ejemplo, el grupo de datos de locación tiene 2 atributos de datos: código y nombre. El método Service Cutter define que el microservicio C persista el atributo de dato “Nombre” y el microservicio D

persiste el atributo de dato “código”. Esto se puede observar en la figura 5-12.

Esto último genera un acoplamiento lógico no deseable entre los dos microservicios. Si se desean actualizar los datos de una entidad, en vez de actualizar la fila de una tabla (como se haría normalemente) se tienen que modificar diferentes filas de multiples tablas de multiples bases de datos. Esta situación no se puede presentar utilizando el método DISC ya que no se contempla que diferentes microservicios persistan un mismo grupo de datos.

5.2. Caso de Estudio: Trading System

Gysel presenta un caso de estudio de un sistema comercial ficticio [14]. Este sistema es diseñado tomando como base los conocimientos que se tienen con otros software de servicio financiero. No existe un documento de requerimientos, o algo parecido, para este sistema, solo un archivo JSON con información básica sobre requerimientos funcionales y no funcionales. De este archivo JSON se obtiene la información necesaria para desarrollar el caso de estudio utilizando el método DISC.

Este sistema cuenta con 10 casos de uso: **Enviar Pedido, Instruir Pedido, Importar Precio, Leer Noticias, Importar Noticias, Ver Recomendaciones, Sugerir Recomendaciones, Crear Cuenta, Crear Cuenta de Dueño, y Ver Portafolio**. Los grupos de datos utilizados por este sistema son: **Dueño de Cuenta, Persona Natural, Compañía, Cuenta, Recomendación, Información de Pago, Instrucción, Orden, Stock, Posición, Precio, Noticias** [14].

En el artículo de Gysel [14] se presenta un método que utiliza algoritmos de agrupamiento para separar un sistema en microservicios. Se busca utilizar la información proporcionada por Gysel como entrada para el método DISC. De manera que se pueda realizar una comparación entre el método de Gysel y el método DISC.

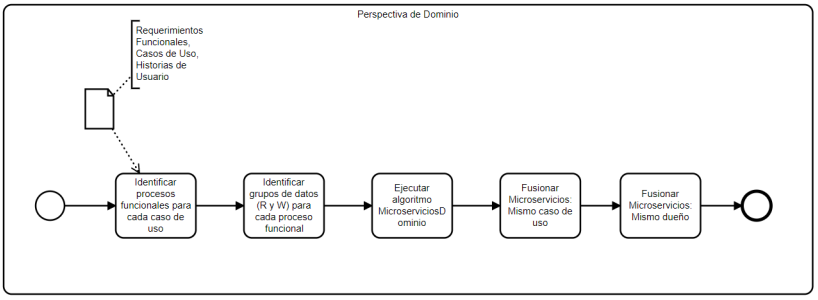

Figura 5-15: Subproceso “Analizar perspectiva de Dominio” expandido.

5.2.1. Analizar Perspectiva de Dominio

Identificar procesos funcionales

La información que se tiene no es suficiente para obtener los procesos funcionales de manera correcta. Sin embargo, los nombres de los casos de uso son suficientemente claros como para generar una aproximación y asumir, para este caso de estudio, que cada uno de estos es un proceso funcional. Lo más adecuado sería tener documentos de requerimientos de software, pero el artículo de Gysel no los incluye. Por tanto, cada uno de los casos de uso se toma en cuenta como un proceso funcional.

Identificar relaciones de lectura y escritura

La información sobre las lecturas y escrituras del sistema se obtienen del artículo de Gysel [14]. Se puede observar esta información en forma de tabla en la figura 5-16. En la figura se observa algo extraño: hay varios grupos de datos que son leídos y no son escritos, o viceversa. Se desconoce la razón de esto. Para fines de este caso de estudio se maneja de esa manera.

Ejecutar algoritmo MicroserviciosDominio

El algoritmo MicroserviciosDominio se encuentra en la sección 4.2.3 de la tesis. Al realizar este algoritmo se obtienen los primero Microservicios del método DISC. Estos microservicios van cambiando mientras avanza el método. Los microservicios obtenidos se pueden observar en la figura 5-17.

Proceso Funcional/Grupo de Datos	GD1 - Dueño de Cuenta	GD2 - Persona Natural	GD3 - Compañía	GD4 - Cuenta	GD5 - Recomendación	GD6 - Información de Pago	GD7 - Instrucción	GD8 - Orden	GD9 - Stock	GD10 - Posición	GD11 - Precio	GD12 - Noticias
PF1 - Enviar Pedido				R				W	R			
PF2 - Instruir Pedido						R	W	W		W		
PF3 - Importar Precio								R	R		W	
PF4 - Leer Noticias								R				R
PF5 - Importar Noticias								R				W
PF6 - Ver Recomendaciones				R	R				R			
PF7 - Sugerir Recomendaciones				R	W				R	R		
PF8 - Crear Cuenta	R	R	R	W								
PF9 - Crear Cuenta de Dueño	W	W	W									
PF10 - Ver Portafolio				R				R	R	R		

Figura 5-16: Relaciones entre procesos funcionales y grupos de datos. W para relación de escritura y R para relación de lectura.

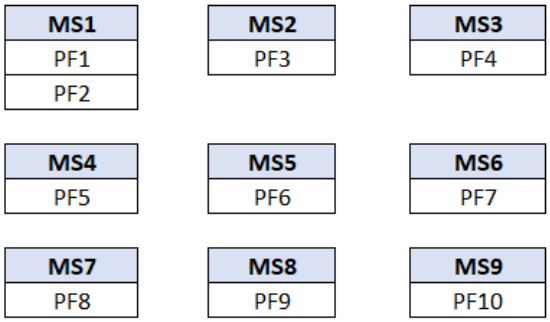

Figura 5-17: Salida de la subtarea “Ejecutar algoritmo MicroserviciosDominio”.

Fusionar Microservicios: Mismo caso de uso

Esta sección se omite porque se toma en cuenta cada caso de uso como un proceso funcional.

Fusionar Microservicios: Mismo dueño

Esta sección se omite porque Gysel no presenta información sobre tipos de usuario.

5.2.2. Analizar Perspectiva de Infraestructura

Separar Microservicios: Diferencia de Almacenamiento

El grupo de datos de Noticias necesita un alto nivel de almacenamiento. Esto porque las noticias incluyen, aparte de texto, imágenes y otros archivos multimedia. Por lo tanto, se clasifica el grupo de datos de Noticias con almacenamiento grande. Con esta información, se clasifica el proceso funcional que se observa en la figura 5-19.

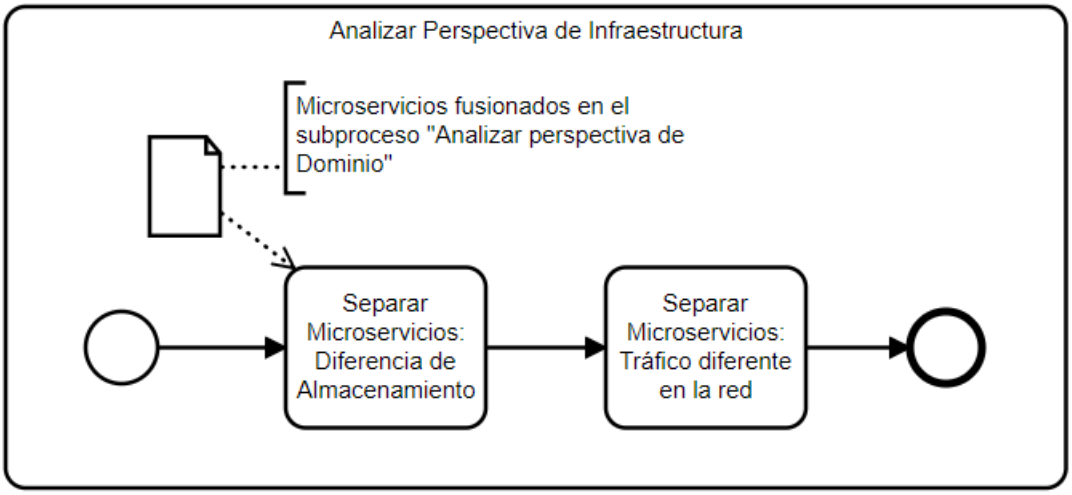

Figura 5-18: Subproceso "Analizar perspectiva de Infraestructura" expandido.

Se puede observar que cada microservicio clasificado tiene un solo proceso funcional, por esto no es posible realizar separaciones.

MS2	Almacenamiento
PF3	Alto

MS3	Almacenamiento
PF4	Alto

Figura 5-19: Clasificación de los procesos funcionales conforme a su nivel de almacenamiento.

Separar Microservicios: Tráfico diferente en la red

Para este caso de estudio no existe la información necesaria para realizar esta tarea.

5.2.3. Analizar Perspectiva de Seguridad

Separar Microservicios: Criticalidad de Seguridad

Los grupos de datos Stock, Noticias y Precio son públicos. Por lo tanto la seguridad de la lectura de los datos en el sistema no es crítica. Al contrario los grupos de datos Dueño de Cuenta, Persona Natural y Compañía son clasificados con seguridad crítica. Con esta información se realiza la clasificación de los procesos funcionales que se observan en la figura 5-21. Se puede observar que cada proceso funcional pertenece a

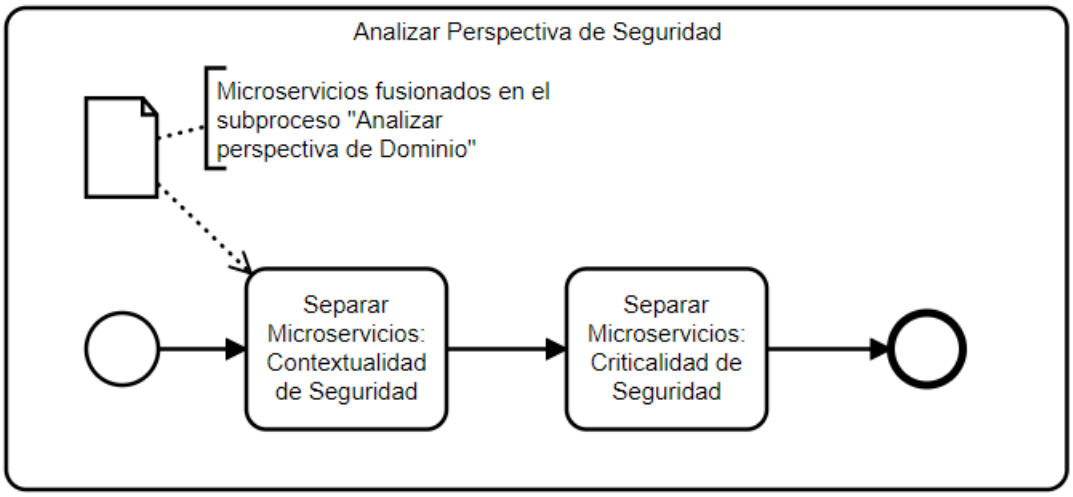

Figura 5-20: Subproceso "Analizar perspectiva de Seguridad" expandido.

un microservicio que no comparte con otros procesos funcionales. Por esto mismo no se realizan separaciones.

MS2	Criticalidad Seguridad
PF3	Baja

MS7	Criticalidad Seguridad
PF8	Alta

MS3	Criticalidad Seguridad
PF4	Baja

MS8	Criticalidad Seguridad
PF9	Alta

MS4	Criticalidad Seguridad
PF5	Baja

Figura 5-21: Clasificación de los procesos funcionales conforme a su criticalidad de seguridad.

Separar Microservicios: Contextualidad de Seguridad

Para este caso de estudio no existe la información necesaria para realizar esta tarea.

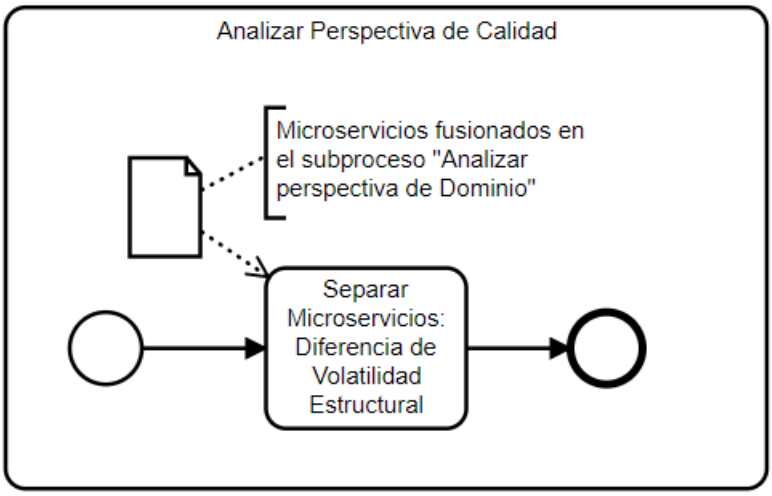

Figura 5-22: Subproceso “Analizar perspectiva de Calidad” expandido.

5.2.4. Analizar Perspectiva de Calidad

Separar Microservicios: Diferencia de Volatilidad Estructural

El grupo de datos “Recomendación” es clasificado como estructuralmente volátil. Con esta información se clasifican 2 procesos funcionales como estructuralmente volátiles. Esto se muestra en la figura 5-23. Se observa que cada proceso funcional pertenece un microservicio que no comparte con otros procesos funcionales. Por esto mismo no se realizan separaciones.

MS5	Volatilidad Estructural
PF6	A menudo

MS6	Volatilidad Estructural
PF7	A menudo

Figura 5-23: Clasificación de los procesos funcionales conforme a su criticalidad de seguridad.

5.2.5. Resolver decisiones contradictorias

Los microservicios que se reciben como entrada para este subproceso se muestran en la figura 5-25.

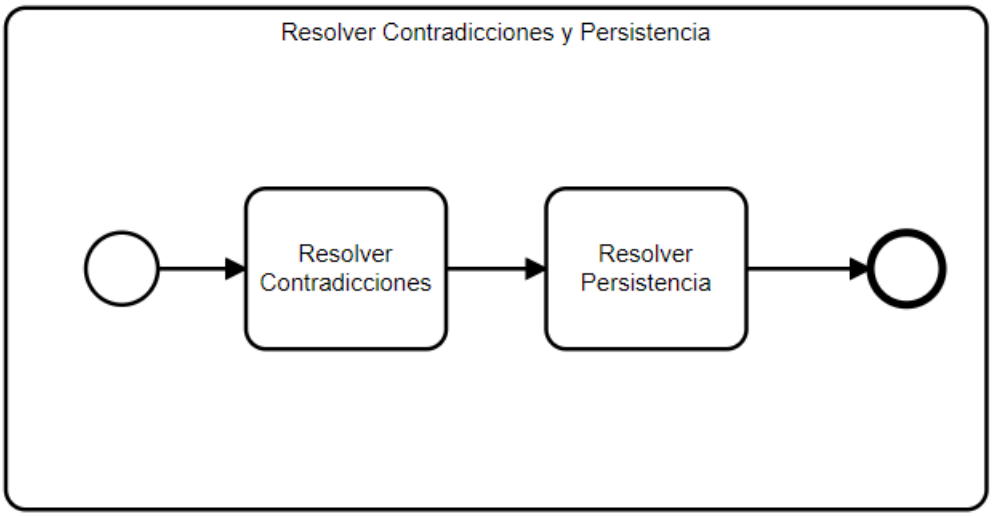

Figura 5-24: Subproceso “Resolver contradicciones y Persistencia” expandido.

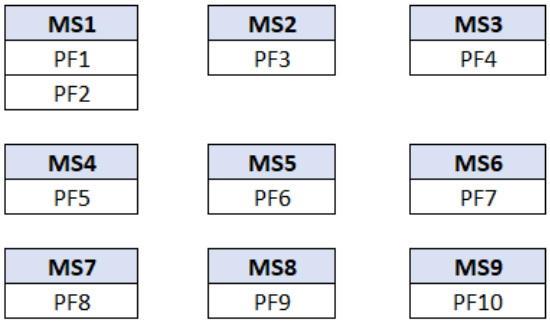

Figura 5-25: Entrada del subproceso “Resolver Decisiones Contradictorias”.

Resolver Contradicciones

Para este caso de estudio no hubo diferencias entre las salidas de los tres subprocesos anteriores (**Analizar perspectiva de Infraestructura**, **Analizar perspectiva de Seguridad** y **Analizar perspectiva de Calidad**). Por esto mismo no hay contradicciones que resolver.

Resolver Persistencia

Como se menciona en la sección 4.6.2 de esta tesis, para tomar estas decisiones se deben tomar en cuenta los siguientes factores: Latencia, Criticalidad de consistencia, Volatilidad estructural y Relaciónes de lectura y escritura entre proceso funcional y grupos de datos.

Se puede dar el caso donde existan diferentes razones para que uno u otro microservicio maneje la persistencia de datos de un grupo de datos. En estos casos se pueden hacer 2 cosas: Priorizar un microservicio o fusionar los microservicios.

Proceso Funcional/Grupo de Datos	GD1 - Dueño de Cuenta	GD2 - Persona Natural	GD3 - Compañía	GD4 - Cuenta	GD5 - Recomendación	GD6 - Información de Pago	GD7 - Instrucción	GD8 - Orden	GD9 - Stock	GD10 - Posición	GD11 - Precio	GD12 - Noticias
PF1 - Enviar Pedido				R				W	R			
PF2 - Instruir Pedido						R	W	W		W		
PF3 - Importar Precio								R	R		W	
PF4 - Leer Noticias								R				R
PF5 - Importar Noticias								R				W
PF6 - Ver Recomendaciones				R	R				R			
PF7 - Sugerir Recomendaciones				R	W				R	R		
PF8 - Crear Cuenta	R	R	R	W								
PF9 - Crear Cuenta de Dueño	W	W	W									
PF10 - Ver Portafolio				R				R	R	R		

Figura 5-26: Relaciones entre procesos funcionales y grupos de datos. W para relación de escritura y R para relación de lectura.

Conociendo la información de la figura 5-26 se puede generar una lista de razones positivas y negativas para que un microservicio maneje o no la persistencia de un grupo de datos. Para este caso de estudio, la lista de razones se puede observar en la figura 5-27. En esta figura, hay una columna llamada "Grupo de datos", donde se clasifican las situaciones a favor o en contra de que un cierto grupo de datos sea persistido por un microservicio. En las fila donde dice $+GDx$ se explica una razón a favor de que ese microservicio persista el grupo de datos x. Al contrario, la fila que dice $-GDx$ se explica una razón en contra de que ese microservicio persista el grupo de datos x. Para este caso de estudio se maneja la clasificación GDx, que representa razones ligeramente a favor de que el microservicio persista el grupo de datos x (principalemente que un microservicio lea el grupo de datos x).

Con esta información se toman las siguientes decisiones:

- **GD1.** Solo MS8 tiene razones para persistir GD1. Por esto MS8 persiste GD1.
- **GD2.** Solo MS8 tiene razones para persistir GD2. Por esto MS8 persiste GD2.
- **GD3.** Solo MS8 tiene razones para persistir GD3. Por esto MS8 persiste GD3.
- **GD4.** Solo MS7 tiene razones para persistir GD4. Por esto MS7 persiste GD4.
- **GD5.** Solo MS6 tiene razones para persistir GD5. Por esto MS6 persiste GD5.

MS1	
Grupo de Datos	**Razones**
GD4	Es leido por PF1
GD6	Es leido por PF2
+GD7	Es escrito por PF2
+GD8	Es escrito por todos los procesos funcionales
GD9	Es leido por PF1
+GD10	Es escrito por PF2

MS2	
Grupo de Datos	**Razones**
GD8	Es leido por PF3
GD9	Es leido por PF3
+GD11	Es escrito por PF3

MS3	
Grupo de Datos	**Razones**
GD8	Es leido por PF4
GD12	Es leido por PF4

MS4	
Grupo de Datos	**Razones**
GD8	Es leido por PF5
+GD12	Es escrito por PF5

MS5	
Grupo de Datos	**Razones**
GD4	Es leido por PF6
GD5	Es leido por PF6
GD9	Es leido por PF6

MS6	
Grupo de Datos	**Razones**
GD4	Es leido por PF7
+GD5	Es escrito por PF7
GD9	Es leido por PF7
GD10	Es leido por PF7

MS7	
Grupo de Datos	**Razones**
GD1	Es leido por PF8
GD2	Es leido por PF8
GD3	Es leido por PF8
+GD4	Es escrito por PF8

MS8	
Grupo de Datos	**Razones**
+GD1	Es escrito por PF9
+GD2	Es escrito por PF9
+GD3	Es escrito por PF9

MS9	
Grupo de Datos	**Razones**
GD4	Es leido por PF10
GD8	Es leido por PF10
GD9	Es leido por PF10
GD10	Es leido por PF10

Figura 5-27: Razones a favor y en contra de que ciertos microservicios persistan ciertos grupos de datos.

- **GD6.** Solo MS1 tiene razones para persistir GD6. Por esto MS1 persiste GD6.
- **GD7.** Solo MS1 tiene razones para persistir GD7. Por esto MS1 persiste GD7.
- **GD8.** Solo MS1 tiene razones para persistir GD8. Por esto MS1 persiste GD8.
- **GD9.** GD9 solo es leído, nunca escrito. Se sabe que los datos de GD9 son públicos, por lo tanto se piensa que son persistidos por una entidad exterior al sistema y, por lo tanto, no hace falta persistirlo.
- **GD10.** Solo MS1 tiene razones para persistir GD10. Por esto MS1 persiste GD10.
- **GD11.** Solo MS2 tiene razones para persistir GD11. Por esto MS2 persiste

GD11.

- **GD12.** Solo MS4 tiene razones para persistir GD12. Por esto MS4 persiste GD12.

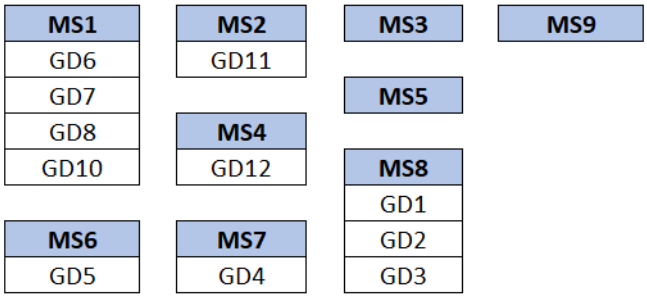

Figura 5-28: Relaciones de persistencia entre microservicios y grupos de datos.

Se ha decidido qué microservicios persistiran qué grupos de datos (Ver figura 5-28). En este caso de estudio es conveniente hacer fusiones de microservicios tomando en cuenta diferentes factores:

- Los grupos de datos escritos por MS8 (GD1, GD2 y GD3) tiene requerimientos de seguridad alta. El proceso funcional de MS7 lee los grupos de datos de MS8. Con el fin de mantener las lecturas y escrituras de los grupos de datos seguros en un mismo microservicio, se fusionan MS7 y MS8.

- Los procesos funcionales de MS5 y MS6 se centran en un mismo grupos de datos (recomendaciones). Lo más probable es que pertenezcan al mismo dueño (recordemos que este caso de estudio no cuenta con información de dueños). Ademas, sus lecturas y escrituras tienen 3 grupos de datos en común. Por lo mismo se decide fusionar estos 2 microservicios.

- Los procesos funcionales de MS3 y MS4 se centran en un mismo grupo de datos (noticias). Lo más probable es que pertenezcan al mismo dueño (recordemos que este caso de estudio no cuenta con información de dueños). Agregado a esto, sus lecturas y escrituras tienen todos los grupos de datos en común. Por todo esto se decide fusionar estos 2 microservicios.

- Los procesos funcionales de MS1, MS5, MS6 y MS9 parecen pertenecer al mismo dominio. Los 4 microservicios tienen funcionalidad referente al portafolio (MS1 maneja pedidos para el portafolio, MS5 y MS6 presenta recomendaciones para el portafolio y MS9 presenta el portafolio en si). Sumado a esto, los procesos funcionales de estos microservicios realizan peticiones a los mismos microservicios (MS5 y GD9). Lo más probable es que pertenezcan al mismo dueño, que sería el dueño del portafolio (recordemos que este caso de estudio no cuenta con información de dueños). Por estas razones, se decide fusionar estos tres microservicios.

Después de estas fusiones, la salida del método DISC se muestra en la figura 5-29.

MS1		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF1	MS5, GD9	GD4, GD9
PF2	MS4	GD5

MS1
Persiste
GD7, GD8, GD10

MS2		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF3	MS1, GD9	

MS2
Persiste
GD11

MS3		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF4	MS1	GD8
PF5	MS1	GD8

MS3
Persiste
GD12

MS4		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF6	MS5, GD9	GD4, GD9
PF7	MS1, MS5, GD9	GD4, GD9, GD10
PF10	MS1, MS5, GD9	GD4, GD8, GD9, GD10

MS4
Persiste
GD5

MS5		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF8		
PF9		

MS5
Persiste
GD1, GD2, GD3, GD4

Figura 5-29: Salida del método DISC.

5.2.6. Comparación del método DISC y el método Service Cutter

El método Service Cutter propuesto por Gysel se explica en la sección 3.3 de esta tesis. El método utiliza algoritmos de agrupamiento para encontrar cortes de servi-

cios. La implementación del método incluye 4 algoritmos de agrupamiento. Dos de estos algoritmos son no deterministas, otro de estos algoritmos recibe como entrada el número de microservicios deseados, el algoritmo restante es llamado Markov. El algoritmo de Markov es determinista y no se recibe como entrada el número de microservicios deseados.

Comparar el método DISC utilizando otro algoritmo diferente al de Markov se vuelve complicado porque, gracias al no determinismo, se obtiene una salida diferente cada vez que se ejecuta el algoritmo, por lo que no se sabe con qué salida realizar la comparación. Lo mismo para el algoritmo en el que se necesita como entrada el número de microservicios que tiene la salida: no se sabe con que número de microservicios realizar la comparación. Por esto, se decide utilizar el algoritmo de Markov para la comparación entre las salidas de Service Cutter y el método DISC.

Salida del método Service Cutter

La salida del método Service Cutter para el caso de estudio de Cargo Tracker es un grafo que se puede observar en la figura 5-30. Además del grafo, el método ofrece información de cada microservicio, un ejemplo de la información que se ofrece se puede observar en la figura 5-31. Con la información de salida que ofrece el método Service Cutter se puede generar un formato simplificado con toda la información relevante, como se muestra en la figura 5-32.

Comparación de las salidas

Para este caso de estudio hubo múltiples diferencias en las salidas: El método Service Cutter tiene 4 microservicios, la granularidad máxima es 6 y acoplamiento máximo es 3. El método DISC tiene 5 microservicios, la granularidad máxima es 3, el acoplamiento máximo es 9 y GD9 no es persistido por ni un microservicio.

El grupo de datos GD9 solo es leído pero nunca escrito. Por esto, la salida del método DISC establece que ni un microservicio se encarga de su persistencia (pensando que es un servicio externo el que ofrece estos datos). Si el método Service Cutter supusiera lo mismo conforme a GD9, su acoplamiento máximo sería 7.

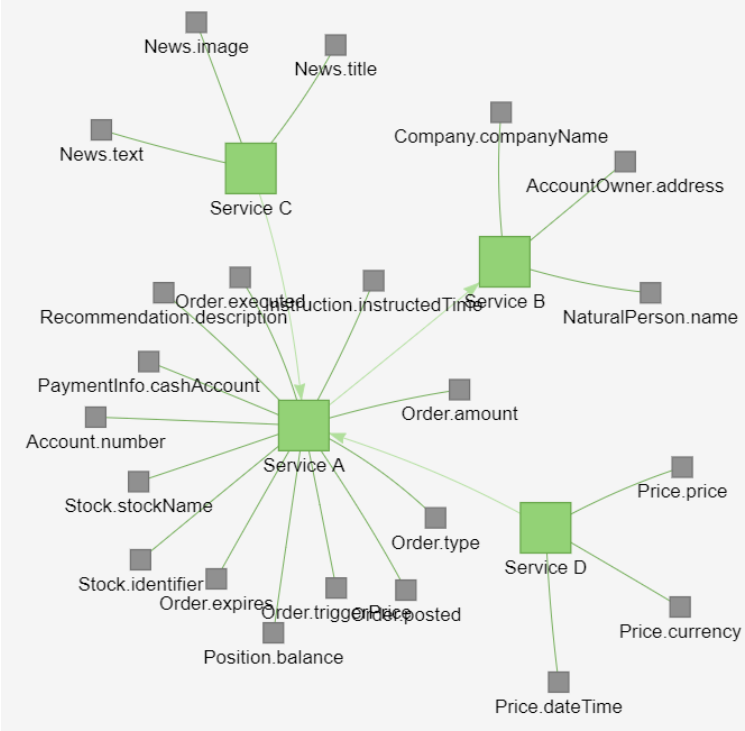

Figura 5-30: Salida del método Service Cutter.

Service A

Responsible for Use Cases:

- PostOrder
- InstructOrder
- View Recommendations
- SuggestRecommendations
- CreateAccount
- viewPortfolio

Published Language

Published Language between **Service A - Service B**

- Company.companyName
- AccountOwner.address
- NaturalPerson.name

Published Language between **Service A - Service C**

- Stock.identifier

Published Language between **Service A - Service D**

- Order.triggerPrice
- Stock.identifier

Figura 5-31: Información que ofrece el método Service Cutter para cada Microservicio.

Para este caso de estudio no se cuenta con mucha información conforme al dominio del sistema, ni que hace cada una de sus funcionalidades. Por esto mismo es

Service A		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF1		
PF2		
PF6		
PF7		
PF8	Service B	GD1, GD2, GD3
PF10		

Service A
Persiste
GD4, GD5, GD6, GD7, GD8, GD9, GD10

Service B		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF9		

Service B
Persiste
GD1, GD2, GD3

Service C		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF4	Service A	GD9
PF5	Service A	GD9

Service C
Persiste
GD12

Service D		
Procesos Funcionales	**Hace peticiones a**	**Grupos de datos de peticion**
PF3	Service B	GD8, GD9

Service D
Persiste
GD11

Figura 5-32: Salida del método Service Cutter con formato simplificado.

complicado saber si dos procesos funcionales, con nombres muy diferentes, pertenecen o no al mismo dominio. Por ejemplo, se desconoce si PF1 y PF2 pertenecen al mismo dominio que PF6, PF7 y PF10 (Se encuentran separados en la salida del método DISC y juntos en la salida del método Service Cutter).

Se considera que, en el servicio A, PF8 no debería estar en el mismo microservicio que el resto de los procesos funcionales. El proceso funcional de crear una cuenta (se piensa que es una cuenta para iniciar sesión en el sistema) debería estar junto con PF9, con el fin de que todo el manejo de cuentas y sesiones se lleve a cabo por un mismo microservicio. Agregado a esto, PF8 hace uso de los datos que son persistidos por el servicio B.

En este caso de estudio la salida del método Service Cutter parece tener mejores resultados en cuanto al acoplamiento entre los microservicios. La situación mencionada sobre GD9 afecta los resultados pero, aun si GD9 no fuera persistido por ni un microservicio, la salida de Service Cutter parece tener menor acoplamiento en términos generales. Se piensa que la falta de información contextual sobre el sistema afecta el resultado del método DISC. Si se conocieran los dueños y el contexto de cada

proceso funcional se podrían tomar mejores decisiones, y probablemente esto llevaría a un mejor resultado. En un proyecto real, cuando se es arquitecto de un sistema, se cuenta más información que la que se tiene para este caso de estudio.

En cuestion de granularidad, para este caso de estudio los microservicios del método DISC tienen menor granularidad. Tiene sentido, para el método DISC, que entre menor sea la granularidad general de los microservicios mayor sea el acoplamiento entre ellos. Menor granularidad de un microservicio, indirectamente se convierte en menor número de grupos de datos persistidos por un microservicio. Esto porque los grupos de datos que se persisten se seleccionan conforme a los procesos funcionales de un microservicio. Entre menos procesos funcionales menos grupos de datos para persistir se seleccionan. Entre menos de grupos de datos se persisten, más comunicación hay con otros microservicios para obtener datos.

5.3. Resumen

En este capítulo se desarrollan 2 casos de estudio utilizando el método DISC. Se realizan las tareas de los procesos y subprocesos, siguiendo el orden del método, para obtener una salida para cada caso de estudio. Las salidas del método DISC se comparan con las salidas del método Service Cutter [14]. Para estos casos de estudio, el método DISC y el método Service Cutter utilizan las mismas fuentes de información como entrada. Por esto último se considera válido realizar la comparación entre los dos métodos.

a

Capítulo 6

Conclusiones

En este capítulo, se presenta un resumen del trabajo propuesto, las conclusiones tomando en cuenta los resultados del Capítulo 5, las contribuciones del trabajo realizado y el trabajo futuro.

6.1. Resumen

Este trabajo de investigación ofrece una propuesta de solución al problema presentado en el capítulo 1, a traves del método diseñado en el capítulo 4. A continuación se hace un análisis de los elementos utilizados para llegar a dicha solución. Se parte de la hipótesis presentada en el Capítulo 1:

Se propone generar un método que ayude a separar un sistema en microservicios. Este método utiliza los requerimientos funcionales a un nivel de granularidad de proceso funcional y los requerimientos no funcionales para tomar decisiones sobre qué procesos funcionales pertenecen a un mismo microservicio, y sobre qué grupos de datos son persistentes por cada microservicio.

El diseño del método que cumple con la hipótesis se puede observar a lo largo de todo el capítulo 4 de esta tesis. Mediante el análisis de los requerimientos funcionales, a un nivel de granularidad de proceso funcional, junto con los requerimientos no funcionales, el presente trabajo introduce un método para separar un sistema en microservicios. El método recibe como entrada los requerimientos funcionales y no

funcionales de un sistema.

Este método se apoya en los criterios de acoplamiento: Proximidad semántica, Dueño compartido, Volatilidad estructural, Latencia, Criticalidad de consistencia, Disponibilidad crítica, Semejanza de almacenamiento, Tráfico similar en la red, Contextualidad de seguridad y Criticalidad de la seguridad. Estos criterios de acoplamiento son mencionados en la literatura como desencadenantes de decisiones en arquitecturas basadas en microservicios.

Añadido a esto, el método propone utilizar conceptos del estándar internacional de medición de tamaño funcional de software COSMIC para medir la granularidad de los microservicios, así como el acoplamiento entre microservicios.

6.2. Contribuciones

Las contribuciones de este trabajo de tesis son las siguientes:

1. **Método con un proceso ordenado:** A diferencia de algunos otros métodos (mas no todos) el método DISC presenta un proceso ordenado que permite separar un sistema en microservicios. El orden de este proceso puede observarse en la sección 4.1 de la tesis, y más a detalle a lo largo de todo el capítulo 4 de la tesis.

2. **Método que toma en cuenta los requerimientos no funcionales del sistema:** Algunos métodos (más no todos) no toman en cuenta los requerimientos no funcionales del sistema. Sin embargo, estos requerimientos deben ser tomados en cuenta al crear la arquitectura de software. El método DISC toma en cuenta algunos de los requerimientos no funcionales de un sistema. Se puede observar que se toman en cuenta requerimientos no funcionales en la sección 4.1 de la tesis, y más a detalle a lo largo de todo el capítulo 4 de la tesis.

3. **Concepto de granularidad de microservicios objetivo:** El concepto de granularidad de microservicios es mencionado múltiples veces en la literatura. Sin embargo, son pocas las definiciones que se pueden encontrar sobre qué

es exactamente la granularidad de microservicios. Las definiciones que se encuentran son subjetivas, de manera que no es posible medir objetivamente la granularidad de un microservicio. En esta tesis se propone utilizar conceptos del método de medición COSMIC para generar una nueva defición de granularidad, de manera que se pueda medir la granularidad de manera objetiva. Este nuevo concepto de granularidad se puede leer la sección 4.7.1 de esta tesis.

4. **Métricas objetivas para microservicios:** Esta tesis presenta una forma de medir granularidad de microservicios y una forma de medir acoplamiento entre microservicios. Ambas métricas se basan en conceptos del método de medición COSMIC. Las métricas propuestas se pueden observar en la sección 4.7 de esta tesis.

5. **Método expandible:** El método DISC no toma en cuenta todos los criterios del catálogo de criterios de acoplamiento presentados en la sección 2.1.3 de esta tesis. Incluso el catálogo del criterios de acoplamiento no contiene todos los criterios que se deben tomar en cuenta. Cada subproceso y tarea que se realiza en el método DISC está marcado de manera clara en los diagramas BPMN del capítulo 4. Por lo que es posible agregar nuevas tareas a un subproceso o nuevos subprocesos. Esto se puede observar a lo largo de todo el capítulo 4.

6.3. Ventajas y Desventajas del método DISC

El método DISC presenta ciertas ventajas y ciertas desventajas sobre algunos métodos.

6.3.1. Ventajas del método DISC

A continuación se expresan las ventajas que tiene el método DISC sobre DDD y sobre los 3 métodos mencionados en el capítulo 3.

1. **Domain Driven Design (DDD).** A diferencia de DDD, el método DISC permite separar un sistema en microservicios siguiendo un proceso ordenado.

Esto ofrece que el practicante de MSA sepa cómo comenzar, los pasos a seguir y cúando terminar. Ademas el método DISC toma en cuenta los requerimientos no funcionales del sistema, cosa que no hace DDD. Otra ventaja del método DISC es que ayuda al arquitecto de software a decidir qué microservicios van a persistir qué grupos de datos.

2. **Requirements Reconciliation for Scalable and Secure Microservice (De)composition.** Este método toma en cuenta requerimientos no funcionales de escalabilidad y seguridad. El método DISC toma en cuenta más requerimientos no funcionales que este método.

   En este método se menciona que hay dependencias entre un requerimiento funcional y otro. Estas dependencias se utilizan en el método, sin embargo no se explica cómo se obtiene el conjunto de requerimientos funcionales ni las dependencias entre estos. El método DISC no tiene problemas en definir el conjunto de requerimientos funcionales, ya que se basa en el concepto de proceso funcional de COSMIC.

   Además se considera que este método no es expandible, ya que las decisiones las toma un algoritmo que contiene un *for* y un *if*. No se cree que haya manera de agregar nueva funcionalidad al algoritmo y, al mismo tiempo, mantener la simplicidad del mismo. El algoritmo tendría que cambiar por completo para agregar nuevas funcionalidades, y se tendrían que realizar casos de estudio para verificar si este nuevo algoritmo funciona. Incluso si el nuevo algoritmo funciona, al momento de cambiar por completo el algoritmo, se considera que el método con el nuevo algoritmo es un método totalmente diferente al original.

3. **Identifying Microservices Using Functional Decomposition.** Este método solo toma en cuenta los requerimientos funcionales del sistema. En cambio el método DISC toma en cuenta requerimientos funcionales y no funcionales.

   Los requerimientos funcionales son separados en *operaciones* y *variables de estado*, pero de igual manera no es claro cómo obtener las operaciones y variables

de estado. El método DISC no tiene este problema al basarse en los conceptos de proceso funcional y grupos de datos de COSMIC.

Una desventaja del método

4. *Identifying Microservices Using Functional Decomposition* es que no es expandible. Está hecho para solo tomar en cuenta los requerimientos funcionales del sistema.

5. **Service Cutter: A Systematic Approach to Service Decomposition.** El método de Service Cutter es el que más se parece al método DISC. Toma en cuenta el mismo catálogo de criterios de acoplamiento, los requerimientos funcionales, y la salida menciona qué microservicios van a persistir qué nanoentidades.

   Una de las ventajas del método DISC es que no se utiliza el concepto de nanoentidad. Nanoentidad es un concepto no estándar, por lo que no es claro qué es una nanoentidad y qué no es una nanoentidad. En vez de eso se utilizan conceptos del método de medición COSMIC.

   Cómo se explica en la sección 3.3 de esta tesis, los cortes que genera Service Cutter se clasifican en 3 categorías: excelente, esperado e irrazonable. Un corte irrazonable es un corte al que no se le encuentra una razón de ser. El método Service Cutter presenta cortes irrazonables en al menos un caso de estudio [14].

   Una ventaja del método DISC es que, por su naturaleza, es imposible que haya cortes irrazonables. El arquitecto, siguiendo el método DISC, toma las decisiones de diseño. Por lo tanto, cada decision de diseño tiene una razón de ser.

   El método DISC también tiene ventajas educativas contra Service Cutter. Service Cutter es una caja negra donde el arquitecto ingresa una entrada y obtiene una salida. Sin embargo, no sabe cómo ni porqué se toman las decisiones (por esto mismo hay cortes irrazonables). El método DISC le permite al arquitecto de software observar cómo su diseño evoluciona y las razones por las cuales evoluciona.

El método de Service Cutter es difícil de expandir. Se tiene que modificar el código que Gysel provee para agregar nuevos criterios de acoplamiento. En cambio, el método DISC es completamente expandible.

6.3.2. Desventajas del método DISC

1. **Conocimiento de COSMIC.** El tema de medición de tamaño funcional de software no es muy conocido en la academia ni en la industria. Para que una persona pueda hacer uso del método DISC tiene que entender varios de los conceptos presentados por COSMIC. Un curso para aprender a medir software con COSMIC puede tomar desde 24 horas hasta 40 horas.

2. **Método más tardado.** Se piensa que obtener una salida con el método DISC es más tardado que obtenerla con 3 de los 4 métodos mencionados en la sección de ventajas. El único que puede ser más tardado es DDD que, al ser un método que no tiene un orden, no se sabe cuánto tiempo se necesita para obtener un resultado. Los otros métodos presentan pocos pasos fáciles de seguir y conceptos más sencillos de entender, o en el caso de Service Cutter, son una caja negra donde solo se ingresa la entrada para obtener un resultado en segundos.

6.4. Trabajo Futuro

Martin Fowler opina en su blog que no hay necesidad de utilizar MSA a no ser que se tenga un sistema demasiado complejo para administrar como un monolito [11]. Es una opinión que el autor de esta tesis comparte. Se considera que los casos de estudio expuestos por esta tesis no son demasiado complejos como para no poder administrarlos como un monolito. Hasta donde se sabe, no hay ni un artículo cuyo tema sea la descomposición de servicios, donde se utilicen sistemas muy complejos para casos de estudio. Sería interesante realizar una comparación del método DISC con otros métodos que separan sistemas en microservicios utilizando sistemas realmente complejos.

En el método DISC se clasifican los procesos funcionales conforme a su almacenamiento, tráfico en la red, seguridad, etc. Las clasificaciones, en su mayoría, se hacen bajo la escala nominal de: alto, medio y bajo. Podría ser interesante convertir esa escala nominal en una escala ordinal mediante el uso de lógica difusa.

Los casos de estudio presentados en esta tesis se llevaron a cabo con información limitada en cuánto a los requerimientos funcionales y no funcionales. Esto se hace así con el fin de poder comparar los resultados con los resultados que presenta el método de Service Cutter. Sería interesante realizar un caso de estudio utilizando un sistema del cual sí se tenga un documento de requerimientos de software y donde se puedan preguntar dudas del sistema de ser necesario. De igual manera, teniendo un documento de requerimientos, se pueden obtener los procesos funcionales y grupos de datos del sistema, a diferencia de los casos de estudio de esta tesis donde se tuvieron que aproximar.

En cuanto a las métricas de granularidad y acoplamiento en microservicios, se piensa que estas métricas cumplen con el modelo contextual de medición de software. Las métricas se basan en conceptos de COSMIC y está demostrado que COSMIC cumple con el modelo contextual de medición de software. Otro trabajo futuro puede ser demostrar que las métricas de granularidad y acoplamiento de microservicios cumplen (o no) con el modelo contextual de medición de software.

Bibliografía

[1] Mohsen Ahmadvand and Amjad Ibrahim. Requirements reconciliation for scalable and secure microservice (de)composition. *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference Workshops, REW 2016*, pages 68–73, 2017.

[2] Mike Amudsen. Learning the three types of microservices. In *Software Architectura Conference*, 2018.

[3] Andrés Carrasco, Brent Van Bladel, and Serge Demeyer. Migrating towards microservices: Migration and architecture smells. *IWoR 2018 - Proceedings of the 2nd International Workshop on Refactoring, co-located with ASE 2018*, pages 1–6, 2018.

[4] Lianping Chen. Microservices: Architecting for Continuous Delivery and DevOps. *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, pages 39–46, 2018.

[5] Common Software Measurement International Consortium (COSMIC). Measurementt Manual v4.0.2. (December):1–115, 2017.

[6] COSMIC Organization. Second Generation - Cosmic Sizing.

[7] Udi Dahan. The Known Unknowns of SOA, 2010.

[8] E Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, volume 7873. Addison-Wesley Professional, 2003.

[9] Josh Evans. Mastering Chaos - A Netflix Guide to Microservices, 2017.

[10] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004.

[11] Martin Fowler. MicroservicePremium, 2015.

[12] Martin Fowler and James Lewis. Microservices - A definition of this new architectural term, 2014.

[13] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. *CEUR Workshop Proceedings*, 2072(October):1–8, 2018.

[14] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9846 LNCS, pages 185–200. Springer Verlag, 2016.

[15] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[16] Irakli Nadareishvili, Roonie Mitra, Matt McLarty, and Mike Amundsen. Microservice Architecture: Aligning Principles, Practices, and Culture. 2016.

[17] Chris Richardson. Microservices Architecture, 2014.

[18] Chris Richardson. *Microservices Patterns*, volume 104. 2017.

[19] Dharmendra Shadija, Mo Rezai, and Richard Hill. Microservices: Granularity vs. Performance. *UCC 2017 Companion - Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*, pages 215–220, 2017.

[20] Jacopo Soldani, Damian Andrew Tamburri, and Willem Jan Van Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[21] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[22] Davide Taibi and Valentina Lenarduzzi. On the Definition of Microservice Bad Smells. *IEEE Software*, 35(3):56–62, 2018.

[23] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[24] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018-Janua(March):221–232, 2018.

[25] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. Identifying Microservices Using Functional Decomposition. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10998 LNCS, pages 50–65. Springer Verlag, sep 2018.

[26] Mark Von Rosing, Stephen A. White, Fred Cummins, and Henk De Man. Business process model and notation-BPMN. *The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM*, 1(January):429–453, 2014.

[27] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. Microservice architecture in reality: An industrial inquiry. *Proceedings - 2019 IEEE International Conference on Software Architecture, ICSA 2019*, pages 51–60, 2019.