
# UNIVERSIDAD DON BOSCO

Vicerrectoría de Estudios de Postgrados

Maestría en Arquitectura de Software

---

## “Propuesta arquitectónica para la migración de un sistema monolítico a un sistema distribuido utilizando domain driven design y microservicios: caso de estudio”

---

**Presentado por:**

Carlos Armando Flores Martínez

José Hernán Arteaga González

José Roberto Bonilla Chávez

ENERO DE 2019

Antiguo Cuscatlán, La Libertad, El SALVADOR, CENTROAMERICA.

# Contenido

## Resumen

En el presente documento está plasmado el trabajo de graduación para optar al grado de máster en arquitectura de software, en la cual se realiza el análisis y evaluación de dos tipos de arquitectura de software que han tenido mucha aceptación en la comunidad informática: la arquitectura monolítica y la arquitectura distribuida en microservicios, tomando como base para este análisis un caso de estudio dentro de una compañía del sector trasporte aéreo. Debido a las políticas de la empresa, su nombre no puede ser mencionado, por lo tanto, se referirá a ella como la compañía, de la misma forma el nombre del software no podrá ser nombrado, por lo tanto, se referirá a él como el sistema. Para mayor contextualización acerca de la empresa, en el capítulo 4 análisis del caso, se detalla acerca de ella.

Específicamente en El Salvador la arquitectura distribuida en microservicios no ha sido incorporada ni explotada en su totalidad por los departamentos de tecnología, que durante este estudio se hará referencia con la siguiente sigla TI (tecnologías de la información), por tal motivo, la información de casos de éxito bajo este tipo de arquitectura es escasa.

El objetivo de este trabajo es proporcionar información sobre el proceso de migración de una arquitectura monolítica a una arquitectura distribuida en microservicios, que sirva como guía para todo el interesado en aplicar un proceso de transformación de este tipo. Para entender el proceso de transformación se abordarán dos conceptos que ayudan a rediseñar aplicaciones con características de ser monolíticas.

El primero es conocido como domain driven design, que de ahora en adelante será mencionado como DDD, este concepto ayuda a diseñar un modelo de dominio partiendo de la lógica del negocio o la problemática que se necesita resolver, de tal forma que se pueda enfocar el desarrollo de manera más eficiente en las actividades y requerimientos que realmente generan valor. Este concepto se profundizará en el próximo capítulo.

Por otro lado, el concepto de arquitectura distribuida en microservicios brinda un enfoque para construir unidades pequeñas de software conocidas como microservicios, que interactúan en un ambiente distribuido, cada uno de estos microservicios es responsable de una funcionalidad de negocio específica y única, de manera que permita hacer un cambio de negocio de forma fácil,

escalable y sin afectar las demás funcionalidades del sistema. Este concepto se profundizará en el próximo capítulo.

Enlazando ambos conceptos, en este caso de estudio (para mayor contextualización del caso de estudio, ver capítulo 4 análisis del caso) el proceso de migración se inicia con el diseño de un modelo de dominio utilizando domain driven design, esto proporciona una estructura bien definida y centrada en las funcionalidades del negocio, la cual se constituye de un modelo de dominio conformado por un conjunto de subdominios, los cuáles deben ser flexibles, independientes y dueños de una única funcionalidad de negocio. Y a partir de este punto se identificarán los microservicios, de forma tal que sean coherentes y con bajo acoplamiento, para finalizar la investigación con una propuesta de arquitectura distribuida basada en microservicios.

# Capítulo 1: Planteamiento general de la investigación

## 1.1 Descripción del tema

***Propuesta arquitectónica para la migración de un sistema monolítico a un sistema distribuido utilizando domain driven design y microservicios.***

En la actualidad la forma en que las empresas hacen negocio ha ido cambiando, debido en gran parte al surgimiento y difusión de la tecnología que forma parte de casi todos los aspectos de la vida cotidiana, se está viviendo una época en donde las empresas más que nunca necesitan adaptarse al constante cambio del mercado para mantenerse en el negocio o simplemente no podrán abarcar terreno y cerrarán sus operaciones.

Con la incorporación del e-commerce surgieron nuevos modelos de negocio y nuevos tipos de transacciones. La computación en la nube y las redes sociales, obligan a que las empresas busquen acercarse a su consumidor final para identificar las nuevas exigencias del mercado, por lo que se hace necesario que las empresas deban adaptar sus operaciones para lograr mantener la competitividad y en el mejor de los casos ganar terreno en un mercado globalizado.

Pero hay que recalcar que para esto las empresas necesitan adaptar sus operaciones bajo la vanguardia del mercado, por lo tanto, las aplicaciones de software que dan soporte a estas operaciones deben evolucionar para ser escalables y mantenible. Bass, Clements y Kazman (2012) para el negocio.

Dentro de los principales factores que impulsan a hacer una refactorización tecnológica a una aplicación de software, es porque muchas veces son sistemas legacy que están adecuados a procesos especialmente elaborados para la operación de la compañía, por lo que el producto no es flexible ni adaptable a los cambios del negocio a pesar de ser sistemas que pueden tener una usabilidad en el negocio e incluso llegar a ser críticos para el mismo Sommerville (2015). La principal desventaja que presentan los departamentos de TI con este tipo de software es que han sido construidas bajo un diseño obsoleto cuyo núcleo es demasiado complejo y acoplado, y en la

mayoría de las ocasiones no aprovechan al máximo los recursos de infraestructura en los que están alojados.

Mediante esta investigación, se busca proporcionar a los departamentos de TI una guía en la cual se logre comprender el proceso del modelado de dominio bajo DDD, desacoplando y abstrayendo la lógica del negocio, y diseñando subdominios los cuales se enfoquen en tareas específicas y únicas, para posteriormente llevarlos a una arquitectura distribuida usando microservicios. De tal forma que exista una base para la transformación de aplicaciones cuya finalidad sea desacoplar su lógica para responder adecuadamente a las necesidades del negocio.

## 1.2 Problemática a abordar

Actualmente en El Salvador, es muy difícil obtener información de empresas que hayan implementado un proceso de rediseño de software utilizando DDD y microservicios (este tipo de documentación puede ser interna de cada empresa, y por políticas de estas no se exponen al público). Por otra parte, dentro de los departamentos de TI la noción de lo que es una arquitectura distribuida no siempre se concibe más allá de una aplicación diseñada en capas que separan los componentes del software (usualmente de 3 capas, presentación, lógica, acceso a datos).

Por esta razón, en este documento se plasma el proceso de transformación de un software monolítico hacia una arquitectura distribuida en microservicios, con base en un caso real dentro de una empresa dedicada a los servicios de transporte aéreo. La problemática se aborda mediante un sistema de software desarrollado dentro de la empresa, que debido al paso del tiempo y a los requerimientos constantes del negocio, su arquitectura inicial se ha vuelto difícil de escalar y mantener, por consiguiente, origina un atraso para adaptar nuevas funcionalidades y responder de forma eficaz a las necesidades de la organización.

El diseño inicial del sistema se basó en una arquitectura monolítica en capas, debido a que en ese momento era suficiente para cubrir las necesidades que tenía la compañía, y además era el estándar que utilizaban para diseñar y desarrollar sus aplicaciones.

Sus procesos de negocio y su procesamiento de datos han sido desarrollados de forma secuencial y síncrona, al igual que el manejo de sus dos fuentes principales de información: una base de datos transaccional y un servicio que provee la información de los pasajeros según los vuelos y su ruta.

En base a métricas de rendimiento hechas por el departamento de calidad de la compañía, se ha comprobado que la aplicación presenta deficiencia en sus tiempos de respuesta, que oscila entre 20 a 30 minutos por vuelo a compensar en horas de alta transaccionalidad (en cada vuelo se puede transportar entre 80 a 160 pasajeros dependiendo la unidad de transporte y la distancia del recorrido). Cuando la afluencia de pasajeros por compensar aumenta, el rendimiento de la aplicación decae ante la cantidad masiva de peticiones que recibe a limites inaceptables (cabe mencionar que la aplicación es utilizada en más de 20 países en América y Europa).

El entorno donde se encuentra el sistema ya no es factible para cubrir las necesidades técnicas y de negocio, debido a que el consumo de los recursos aumenta según la demanda y esto tiene una repercusión directa en los tiempos de respuesta.

### 1.3 Antecedentes

La afluencia de clientes que atiende la compañía ha aumentado significativamente, por lo que muchas aplicaciones se han visto en la necesidad de migrar a nuevas tecnológicas y arquitecturas. La aplicación analizada en este caso de estudio fue desarrollada en el año 2011, como respuesta a un proceso de expansión y fusión entre dos compañías, adicional a esto se extendió el uso de esta aplicación a nuevas áreas de negocio, ha llegado a ser utilizada en más de 10 países, y todo esto ha elevado considerablemente la carga de transacciones que debe atender.

De la distancia recorrida entre un punto a otro depende el tipo aeronave, y de esta a su vez depende la cantidad de pasajeros a movilizar, pero se registra un promedio de 130 pasajeros por viaje. Si un viaje por alguna razón no se puede realizar, la compañía se ve obligada a compensar a todos los pasajeros ya sea con beneficios de hoteles, alimentación, transporte, millas, efectivo. Según los datos proporcionados por la compañía, existen diferentes motivos por

los que un viaje no se puede realizar, por ejemplo: mal clima, desperfectos mecánicos, sobreventa de asientos, cierre de aeropuertos.

La afectación que tiene el área de negocio es que el sistema de software no puede procesar de forma eficaz la gran cantidad de transacciones a la que es sometida, y esto eleva considerablemente el riesgo que se generen demandas, por el incumplimiento de las condiciones de compensación que ha establecido la compañía.

Unas de las razones que ha provocado el mal rendimiento del sistema de software, es que fue diseñado sin considerar la escalabilidad de este, ni el aumento en la carga transaccional. Por consiguiente, a lo largo del tiempo todo requerimiento fue desarrollado bajo el mismo núcleo, lo que se ha provocado un gran acoplamiento en su lógica.

Al tratarse de una aplicación monolítica, todo el procesamiento se ejecuta en el mismo hardware, y lo que dificulta aún más el entorno de su implementación, es que hay más aplicaciones alojadas en el mismo servidor, por lo que el uso de recursos compartidos es otro de los problemas detectados.

Para remediar dicho problema, la aplicación ha sido implementada bajo una arquitectura de servidores distribuida horizontalmente, es decir, existe una instancia de la aplicación monolítica en dos servidores, y se utilizan balanceadores de carga para redirigir las transacciones. Pero esta forma de despliegue no soluciona el problema de escalabilidad, como se ha mencionado es necesario responder de forma ágil a los requerimientos de negocio.

A partir de esta problemática surgió la necesidad de descomponer la lógica en un modelo de dominio, en el cual cada proceso de negocio sea independiente, de tal forma que pueda migrarse a una arquitectura distribuida, donde no dependa del mismo hardware para procesar las peticiones, sino que la carga de transacciones pueda ser distribuida en la red, es decir, una escalabilidad vertical, según Lafuente (2012).

Actualmente se ha finalizado el proceso de definición de dominio par el sistema utilizando DDD con el objetivo de identificar el dominio y separarlo de todos los componentes de la aplicación, aunque siga utilizando una arquitectura monolítica. El procesamiento de la información se ha desarrollado de forma asíncrona de manera que la consulta de datos no aumente el tiempo de respuesta. Cada uno de estos componentes se construyó mediante capas aisladas porque son menos costosas de mantener y pueden cambiar en un momento dado sin afectar a las demás capas.

A partir de la existente definición del dominio se llevará a cabo el diseño de una nueva arquitectura distribuida para el sistema.

## 1.4 Justificación

Las tecnologías a lo largo del tiempo se van actualizando, sin embargo, quedan desfasadas con celeridad, las necesidades del negocio van cambiando y creciendo con el objetivo de satisfacer al usuario final y brindarle una experiencia sin igual. Debido a ello, los departamentos de tecnología se ven obligadas continuamente a realizar cambios en sus sistemas de forma ágil para generar valor y respuesta lo más pronto posible. Sin embargo, muchas empresas se ven impactadas debido a las necesidades cambiantes del negocio, y a la falta de agilidad para adaptar sus sistemas de información a estas.

Como se ha mencionado, la aplicación analizada en este caso de estudio fue construida bajo una arquitectura monolítica, ésta cumplía con sus requerimientos, sus necesidades, y su visión en ese momento, sin embargo, con el crecimiento que ha tenido la compañía, el área de negocio ha requerido de nuevas funcionalidades, por lo que esta arquitectura ya no es adecuada y no responde a las nuevas exigencias. Es por ello por lo que se inició la construcción de un nuevo diseño arquitectónico bajo una arquitectura distribuida para acelerar los despliegues, hacer cambios sin afectar otras funcionalidades de la aplicación, brindar alta disponibilidad, y no dar de baja a toda la aplicación cada vez que se realice un cambio sobre la misma.

Es importante analizar el contexto de la aplicación, lo cual ayuda a comprender el problema y permite generar una alternativa de solución. Además, de esta investigación se creará una guía de referencia general para migrar una aplicación monolítica hacia una distribuida aplicando los conceptos de DDD y microservicios.

Utilizar una arquitectura de microservicios sugiere una gran ventaja competitiva para las empresas, puesto que faculta a los departamentos de TI a construir aplicaciones que respondan con rapidez a sus necesidades, y con esta investigación se busca apoyarles para aumentar su nivel de madurez en cuanto a la integración de sus sistemas y así mejorar su acuerdo de nivel de servicio para con sus clientes**.**

Finalmente, y no menos importante, los temas de arquitectura de microservicios y DDD están en la mente de cada persona que ejerce esta profesión de ciencias de la computación, sin embargo, no todos saben cuál es el camino para implementarlo, de dónde partir, y muchas veces únicamente quedan los conceptos e ideas a nivel de papel, y difícilmente pueden ser implementados en su día a día. Es por ello que se busca en esta investigación proveer a personas informáticas una guía para poder entender el paso a paso de cómo llevar a cabo esta implementación en un caso de estudio real, y salir de la teoría llevando a la práctica los conceptos que muchos manejan al respecto.

## 1.5 Objetivos

### 1.5.1 Objetivo general

Diseñar una propuesta de arquitectura distribuida para una aplicación monolítica partiendo de su modelo de dominio existente utilizando el enfoque de domain driven design y microservicios.

### 1.5.2 Objetivos específicos

1. Identificar los componentes del dominio que se migrarán a microservicios distribuidos.
2. Definir una arquitectura distribuida usando microservicios partiendo del modelo de dominio.
3. Elaborar un documento de recomendaciones para la migración de un sistema monolítico a un sistema distribuido.

## 1.6 Alcances

Construir una guía de referencia que permita orientar a una persona con conocimientos en ciencias de la computación a llevar a cabo la migración de una aplicación construida en una arquitectura monolítica a una arquitectura distribuida utilizando domain driven design y microservicios. Entregar una propuesta de arquitectura distribuida para la aplicación del caso de estudio.

## 1.7 Limitantes

En este caso de estudio no se tomará en cuenta la migración e implementación de la propuesta arquitectónica distribuida de la aplicación.

No se especificará ninguna tecnología sobre la cual se deberá desarrollar o implementar cada componente que forme parte de la propuesta arquitectónica.

No se detallará ningún componente externo con el cual se comunica la aplicación analizada en el caso de estudio.

## 1.8 Preguntas de investigación

1. ¿Cómo fue el proceso de descomposición del dominio usando domain driven design?
2. ¿Cuál diseño de arquitectura se propone para solventar la problemática actual de la aplicación?

# Capítulo 2: Marco conceptual

Para poder entender a profundidad la investigación plasmada en este documento, es necesario aclarar los términos y conceptos base. Es importante comprender la terminología de la cual se hablará en el desarrollo del documento, y para ello se hará uso de diferentes fuentes bibliográficas que ayuden a definir los siguientes conceptos dentro de este estudio.

## 2.1 Dominio del software

Evans (2003) en su libro "Domain Driven Design" define el dominio como el corazón de la aplicación, debido a que el dominio refleja el comportamiento y el modelo de la aplicación, en el cual se definen los procesos y las reglas de negocio sobre las cuales ha sido construida la aplicación. Prácticamente, Evans (2003) hace referencia al dominio a todo aquella parte del sistema que se puede hablar con un experto en el negocio, es decir, la lógica sobre la cual ha sido desarrollada y que, sin importar la tecnología, el experto pueda entenderla. Haywood (2009) define el dominio de la aplicación como el modelo de clases o responsabilidades de las clases que conforman el software. Esto puede complementarse con la definición con la de Vernon (2013) quien define el concepto de dominio de una empresa como la forma en que esta lleva a cabo sus operaciones. Por tanto, se puede decir que el dominio de software es la forma en cómo el software lleva a cabo la funcionalidad(es) para la cual(es) fue concebido y si toda esta información se combina con los enfoques de los autores Sommerville (2015) que presenta que el modelo de dominio es en sí el modelo de clases de una aplicación o una representación a partir del diseño de la aplicación de lo que un software es. En esta investigación se entenderá el concepto de dominio de software como: la capa de negocio que representa la funcionalidad del software, la cual es independientemente a la tecnología sobre la que se construye, y tiene la facilidad de comprensión para el experto del negocio.

## 2.2 Domain driven design

Teniendo una definición clara del dominio, se puede comprender su importancia a la hora de diseñar el software. Para entender el contexto donde se utiliza DDD hay que analizar primeramente la problemática que se busca solucionar. El problema real al que se enfrentan muchas empresas es que sus sistemas no presentan algún tipo de arquitectura distinguible, lo cual se hace notorio cuando al momento de desarrollar una mejora o un cambio en el flujo de trabajo, se convierte en un desafío debido a la complejidad de entender y modificar el código. Tales sistemas son descritos como "código que hace algo útil, pero sin explicar cómo" Millett (2015), página 4, y mucho peso tiene la complejidad del dominio mezclada con las complejidades técnicas. Evans, (2003).

Este concepto fue introducido por Eric Evans en su libro: Domain Driven Design: Tackling Complexity in the Heart of Software (Addison-Wesley Professional, 2003), y lo define como "una forma de pensar y un conjunto de prioridades, encaminadas a acelerar proyectos de software que tienen que lidiar con dominios complicados.". Evans (2003), página 14

Según Millett (2015) DDD se ocupa de la necesidad de entender el dominio del problema y modelar[1] la arquitectura de una aplicación, con la finalidad de gestionar eficazmente la construcción y el mantenimiento del software.

DDD provee herramientas de modelado estratégico y táctico necesarias para diseñar software de alta calidad, que cumpla con los objetivos de negocio Vernon (2013).

Para poder implementarlo, es necesario que el equipo de desarrollo tenga una estrecha comunicación con los expertos del dominio (que muchas veces resultan ser los usuarios finales) para generar un lenguaje ubicuo[2]. Este cruce de conocimientos produce una descomposición de un dominio problemático en subdominios más manejables y al mismo tiempo se identifica el subdominio central, que es la razón de ser del software. Evans (2003) y Millett (2015).

---

1 Modelo: Un modelo es una forma de conocimiento selectivamente simplificada y conscientemente estructurada. Un modelo apropiado hace sentido de la información y se enfoca en un problema. Evans (2003).

2 Lenguaje ubicuo: Es un lenguaje de equipo compartido. Es usado y compartido por todos en el equipo del proyecto. Vernon (2013).

A partir de las definiciones anteriores, se concluirá que: DDD es un enfoque para el desarrollo de software, cuya finalidad es la creación de un modelo partiendo de un dominio complejo, el cual ayuda a los desarrolladores, expertos del dominio y todos los involucrados, a crear un lenguaje ubicuo que les permita comprender los aspectos del negocio y así generar un software mantenible y de calidad.

Cuando se usa DDD, se tiene la ventaja que no se requiere de una arquitectura específica, dado que el subdominio central reside en un contexto limitado, es decir, que dependerá de las demandas que se esperen del sistema, el uso de patrones y estilos arquitectónicos. Lo que se debe tener en cuenta es que la arquitectura seleccionada debe soportar el aislamiento de la lógica de dominio del resto de las complejidades técnicas. Vernon (2013) y Millett (2015).

### 2.2.1 Arquitectura de domain driven design

A continuación, se abordará la arquitectura de DDD expuesta por Evans (2003) la cual está compuesta por las siguientes capas: capa de presentación, capa de aplicación, capa de dominio y capa de infraestructura.

- **Capa de presentación**

Según la definición de Evans (2003), la capa de presentación es la responsable de mostrar la información e interactuar con el usuario. El actor externo puede ser bien una persona o bien un sistema informático.

La capa de presentación o interfaz de usuario sólo debe contener el código referente a mostrar y recibir datos del usuario. Es importante que esta capa no contenga ningún tipo de lógica de dominio (negocio). Puesto que el usuario final puede ser un sistema informático, esta capa podría proporcionar los medios necesarios para invocar remotamente un api en forma de un Servicio de Host Abierto[3] Vernon (2013).

3 Servicio de host abierto: Define un protocolo que da acceso a un subsistema como un conjunto de servicios. Se abre el protocolo para que todos los que necesitan integrarse a los componentes puedan utilizarlo. Evans (2015). (Evans, 2015).

Por lo tanto, en esta investigación se definirá la capa de presentación como la responsable de mostrar información al usuario, y la encargada de interactuar con un agente externo que puede ser una persona o un sistema de información; y que en su lógica no existan componentes ni validaciones de negocio.

Los componentes de la interfaz de usuario son clientes directos de la capa de aplicación; por lo tanto, debemos comprender el nivel de interacción entre la capa de presentación y la capa de aplicación.

- **Capa de aplicación**

La capa de aplicación según Evans (2015), define los trabajos que el software debe hacer y dirige los objetos del dominio para poder dar solución a los problemas. Las actividades de esta capa son de gran importancia para el negocio y para la interacción con las capas de aplicación de otros sistemas. Al igual que la capa de interfaz de usuario, no debe contener reglas o conocimientos de dominio. Esta capa, se encarga de coordinar tareas y delegar trabajo a los componentes de la capa de dominio, y su estado puede reflejar el progreso de una tarea para el usuario u otro programa.

Millett (2015) define esta capa como la representación de los casos de uso y el comportamiento de la aplicación. Dichos casos de uso se pueden implementar como servicios de aplicación que coordina el cumplimiento de un caso específico, delegando el trabajo al dominio y las capas de infraestructura. Los servicios de la capa de aplicación se construyen con un nivel superior al de los objetos de dominio, por lo que ocultan los detalles de la capa de dominio.

Por lo tanto, se concluye que la capa de aplicación es creada sin ninguna lógica de dominio, y es la orquestadora del trabajo necesario que deben realizar los objetos de dominio, con el fin de solucionar un problema o, en otras palabras, un caso de uso.

Los servicios de aplicación de esta capa son clientes directos del modelo de dominio, por lo que se necesita definir el concepto de la capa de dominio.

- **Capa de dominio**

La importancia de esta capa es fundamental para el enfoque de DDD, y es la responsable de representar los conceptos del negocio. Esta capa es el corazón del software. Evans (2003).

Es el área de código que contiene el modelo de dominio, y aísla las complejidades del modelo respecto a las funcionalidades técnicas de la aplicación, es decir, no depende de ninguna otra capa. Para interactuar con las capas de aplicación e infraestructura, la capa de dominio expone interfaces, las cuales son implementadas de modo que la dependencia del dominio se hace sin acoplamiento Vernon (2013).

Bajo estos conceptos se definirá que, la capa de dominio es la que se ocupa de las necesidades del negocio, sus componentes implementan la funcionalidad del software y su principal razón de ser es separar las reglas del dominio de las complejidades técnicas.

Es claro que los componentes del dominio necesitan algún tipo de almacenamiento para guardar sus estados e información, lo cual hace que la capa de dominio sea cliente directo de la capa de infraestructura para el manejo de la información.

- **Capa de infraestructura**

Retomando la definición de Millett (2015), la capa de infraestructura provee la capacidad técnica que soportan a las capas superiores, lo cual le permite a la aplicación funcionar. Esta capa se enfoca en las capacidades puramente técnicas, tales como: el acceso y almacenamiento de datos, permitir que la aplicación pueda ser consumida por una interfaz de usuario, por aplicaciones a través de servicios web o terminales de mensajes, entre otros. Estos detalles técnicos no deben afectar directamente la lógica de dominio de una aplicación.

Por lo tanto, para este caso de estudio se podrá decir que, la capa de infraestructura es la encargada de proveer los componentes técnicos necesarios para que el software funcione. Su lógica debe estar desacoplada de las otras capas, de tal manera que no afecte directamente a los componentes del dominio.

### 2.2.2 Principio de inversión de dependencia

Luego de entender cada capa involucrada en la arquitectura de DDD, es necesario profundizar en las técnicas de desacoplamiento de capas que están basadas en el principio de inversión de dependencia, el cual fue postulado por Martin y Micah (2006) en la página 201; su definición formal establece que: “Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles. Los detalles (implementación) deben depender de las abstracciones”

La comunicación que proporcionan las capas de bajo nivel (infraestructura), deben depender de las interfaces (abstracciones) definidas por los componentes de alto nivel (presentación, aplicación y dominio), Vernon (2013).

Su propósito es poder crear capas de alto nivel desacopladas e independientes de las implementaciones de las capas de bajo nivel, de forma que sea posible la reutilización los mismos componentes de alto nivel con diferentes implementaciones de componentes de bajo nivel, C. Torres, U. Castro, M. Barroso, J. Calvarro, (2010).

Ahora que se ha definido el concepto de DIP, se podrá entender para el presente trabajo que, el Principio de Inversión de Dependencia, tiene como propósito el desacoplar las dependencias directas entre las capas de alto nivel y las capas de bajo nivel, por medio de interfaces que permitan de forma indirecta la accesibilidad de tipo top-down, permitiendo la reutilización de componentes de alto nivel.

Con el concepto definido sobre DDD y su diseño básico de arquitectura en capas, se ha logrado explicar cómo se pueden abordar los problemas de dominio complejos. Lo importante, es comprender que este concepto no impone una arquitectura en particular, sino que se puede adecuar este enfoque a diferentes tipos de arquitectura.

Por lo tanto, el próximo paso es comprender la importancia que tiene un buen diseño arquitectónico antes de desarrollar los componentes del sistema, de esto depende en gran medida el éxito o el fracaso del producto final.

## 2.3 Arquitectura de software

Para poder llegar a un concepto de arquitectura que se pueda utilizar para el propósito del presente trabajo, se retomará la definición de Coulouris, Dollimore, Kindberg y Blair (2012), página 8 en la cual expresan que: “La arquitectura de software, es la estructura de los componentes separados de un sistema de software y sus interrelaciones. Su propósito es asegurar que dicha estructura cumplirá con los requerimientos presentes y futuros que se requieran de esta”, también Coulouris, Dollimore, Kindberg y Blair (2012) página 110 agrega que: “Los componentes de Software en sistemas distribuidos pueden ser desarrollados e implementados independientemente”

Esta definición es amplia un poco más por Reynoso (2004) en la página 12, que define: "La arquitectura de software es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión.”.

Estas definiciones concuerdan que la arquitectura de software es la estructura de los elementos que conforman un sistema, y la forma de cómo estos elementos están relacionados entre sí para lograr el propósito. También podemos entender que un sistema de software está compuesto de:

Componentes: Los componentes de un software deben entenderse como objetos abstractos, pueden representar un subsistema, un componente o herramienta de software o, dicho de otra forma, un elemento de software cuyo significado y forma puede variar dependiendo del sistema de software.

Propiedades: Se refiere a las características que presenta cada uno de los componentes y como estas pueden condicionar de cierta manera las relaciones entre estos.

Relaciones: es la forma en como los componentes o elementos de un software están relacionados entre sí, la forma en como estos se relacionan viene dada por la forma que tienen de intercambiar mensajes para lograr total o parcialmente el objetivo del software.

Por lo tanto, la arquitectura muchas veces va orientada a solventar uno o varios aspectos del problema que se desea resolver según las conclusiones de Stephens (2015).

En base a estas definiciones, se puede decir que, la arquitectura de software es la disciplina que trata de ordenar los diferentes componentes de un sistema, mediante sus relaciones, con la finalidad de crear una estructura funcional adecuada para cumplir con la funcionalidad esperada.

Independientemente del tipo de arquitectura que se utilice, los requerimientos funcionales podemos representarlos como unidades funcionales que son construidas en la arquitectura utilizando al menos un patrón de diseño que se ajuste a las necesidades de diseño. Es por ello que entender el concepto de unidad funcional y patrón de diseño, es fundamental para llevar a cabo el desarrollo de esta investigación.

### 2.3.1 Unidad funcional

Según Bachmann, Bass, Clements, Garlan, Ivers, Little, Nord y Stafford (2011) se puede definir una unidad funcional como la implementación de una pieza de software que provee una funcionalidad coherente desde el punto de vista modular de la aplicación, es decir que una unidad funcional es aquella porción de software que en el modelo de éste representa una funcionalidad única del mismo.

Por otro lado, los autores tanto como Sommerville (2015) y Stephens (2015), concuerdan con que el modelo del software se construye partiendo de la funcionalidad de este, es decir de los requerimientos funcionales. Por tanto, dentro de esta investigación se entenderá el concepto de unidad funcional como un módulo de la aplicación que se construye a partir de las especificaciones de al menos un requerimiento funcional del sistema.

### 2.3.2 Patrón de diseño

Gamma, Helm, Johnson y Vlissides (1994), página 12 expresa que: “Un patrón de diseño describe un problema que pasa una y otra vez dentro de nuestro ambiente, y que describe el núcleo de la solución a ese problema de tal manera que dicha solución pueda ser utilizada millones de veces sin necesidad de aplicarla exactamente de la misma manera”; es decir, que un patrón de diseño puede utilizarse múltiples veces para solucionar múltiples problemas similares.

Para Haywood (2009) los patrones de diseño son soluciones a problemas de diseño comunes y son útiles para brindar un cierto nivel de abstracción a dicho problema. Sin embargo, el autor Evans (2015) indica que los patrones de diseño se utilizan dependiendo del tipo de dominio de la aplicación descrita, pero no descarta la posibilidad de que puedan utilizarse patrones de diseño que no se adapten perfectamente al dominio descrito, es decir, que ciertos patrones de diseño se adaptan para diseñar cierto tipo de dominios.

Adicionalmente, se puede decir que de los 21 patrones de diseño que propone Gamma, Helm, Johnson y Vlissides (1994), estos tienen la intención de solventar ciertos tipos de problemas de diseño según como se describa el dominio y arquitectura de la aplicación. E. Gamma, Helm, Johnson y Vlissides (1994) propone una tipología para los patrones de diseño, donde los clasifica como creacionales, estructurales y de comportamiento, esta misma tipología la comparte con Allen Holub (2004) y como el nombre de cada categoría lo indica, el propósito de cada patrón de diseño identificado en cada grupo obedece tanto a la forma de crear objetos, la forma como los objetos son agrupados accedidos y la forma en cómo se relacionan entre sí.

Los patrones creacionales agrupan aquellos patrones que definen la forma como se crearan los elementos del sistema o subsistema donde se utilicen, Gamma, Helm, Johnson y Vlissides (1994), página 94 propone que este tipo de patrones son muy útiles para ser implementados en sistemas donde se espera una expansión en la funcionalidad al expresar: “Los patrones creacionales se vuelven importantes para los sistemas que evolucionan integrando más objetos a la herencia de clases”.

Los patrones estructurales se refieren a la forma de como los elementos del sistema son ordenados entre sí para ser utilizados, Gamma, Helm, Johnson y Vlissides (1994), página 22 expresa que este tipo de patrones “Describen formas para organizar los objetos para obtener nuevas funcionalidades” mientras que Allen Holub (2004) se limita a decir que los patrones de diseño estructurales son todos aquellos que tienen que ver con la forma de cómo están organizados los objetos en un programa.

La última categoría propuesta por Gamma, Helm, Johnson y Vlissides (1994) son los patrones de diseño de comportamiento, los cual se refieren al tipo de comportamiento que tendrán los elementos del programa para llevar a cabo su funcionalidad, Allen Holub (2004),

página 377 se refiere a este tipo de patrones como “Los patrones que defines los roles de los objetos y la forma en que estos objetos interactúan con otros objetos”. Esta clasificación de patrones de diseño nos da una mejor idea de qué tipo de patrones de diseño se deben utilizar para resolver ciertos patrones de funcionalidad de la arquitectura(s) que se utilice.

Como señala Evans (2003) la arquitectura de un sistema está orientada a solventar los problemas de diseño y las necesidades de la aplicación; Por lo tanto, se concluye que para este estudio se entenderá como patrones de diseño a la forma de describir el dominio de una unidad funcional de la aplicación que esté orientado a solventar los problemas de diseño de la aplicación y asimismo tienen relación al tipo de arquitectura de la aplicación, es decir, que existen patrones de diseño que se ajustan mejor que otros cuando se trata de una arquitectura monolítica o distribuida, los cuales se desarrollarán a continuación.

### 2.3.3 Tipos de arquitectura de software

Originalmente se consideraba a la construcción de aplicaciones un arte debido a que resultaba demasiado complejo para muchos el entender cómo es que se construyen las estructuras de dato y las iteraciones entre unidades de funcionamiento lógico para formar las primeras aplicaciones de software, pero con el tiempo se desarrollaron nuevas formas de entender y aplicar estas técnicas de desarrollo abstractas que funcionan como guías para resolver problemas de construcción de software, este conjunto de técnicas se les ha llamado arquitectura de software, así podemos decir que la arquitectura es un nivel de diseño que hace énfasis en los aspectos que se encuentran más allá de los algoritmos y estructuras de datos, es el diseño y especificación de la estructura global del sistema, es un nuevo tipo de problema .Garlan y Shaw (1994)

Habiendo entendido dichas definiciones, es muy importante abordar dos conceptos para el desarrollo de este caso de estudio. El primer concepto es acerca de uno de los primeros diseños de arquitectura desarrollados para sistemas empresariales: arquitectura monolítica donde la aplicación está constituida por un único programa de software que hace todo. Stephens, (2015); y el segundo concepto es acerca de la evolución de la arquitectura tradicional a un ambiente distribuido y escalable: arquitectura distribuida que podemos definir como un conjunto de

programas interactuando en una red de computadoras que parecen ser un programa único. Coulouris, Dollimore, Kindberg y Blair (2012).

- **Arquitectura monolítica**

Como se ha definido anteriormente, al hablar de arquitectura de software se hace referencia a los componentes y sus relaciones. Ahora, abordando el concepto de arquitectura monolítica de Stephens, (2015) en la página 96, expresa que: "En una arquitectura monolítica, un simple programa hace todo, muestra la interfaz de usuario, acceso a los datos, procesa las órdenes de compra, imprime facturas, lanzamisiles y hace todo lo demás que la aplicación necesite hacer". También Stephens (2015) señala que una arquitectura monolítica está compuesta por piezas funcionales altamente acopladas.

De manera similar, Lenz y Wienands (2006), que expresa que una arquitectura monolítica consta generalmente de muchos componentes diferentes, estos bloques de construcción están fuertemente interconectados, no uniformes y entretejidos, lo cual dificulta la separación de bloques individuales del resto del sistema, por lo tanto, los hace prácticamente inutilizables en el contexto de otros sistemas. Este acoplamiento se ve reflejado en dependencias de largo alcance y alta complejidad.

La complejidad se puede observar al transcurrir el tiempo, cuando los sistemas con una arquitectura monolítica necesitan mantenerse, ampliarse y adaptarse a las necesidades del negocio. Las consecuencias de las arquitecturas monolíticas, las encontramos cuando el cambio en una parte del sistema provoca el cambio en otras, provocando que los mantenimientos sean extensos, engorrosos y costosos, según Lenz y Wienands (2006).

Para ello, se definirá la arquitectura monolítica como una arquitectura donde todos los componentes del software se encuentran altamente acoplados, y se alojan en un mismo bloque, lo cual provoca una complejidad al momento de realizar mejoras sobre la aplicación y así escalar su funcionalidad.

Al pasar el tiempo, la evolución de las arquitecturas de software tuvo un cambio para diseñar aplicaciones que fuesen robustas, escalables, mantenibles y desacopladas. Por lo tanto, se

requiere definir el concepto de arquitectura distribuida que es necesario para el entendimiento de esta investigación y solución de esta.

- **Arquitectura distribuida**

Los autores Tanenbaum y Steen (2006) en la página 20, definen una arquitectura distribuida como "una colección de computadoras independientes que parece para sus usuarios un sistema coherente único". Cabe resaltar en esta definición que en este tipo de arquitectura cada componente del sistema es totalmente autónomo; además, como cada componente es independiente y forma parte de un todo, eso significa que cada parte debe colaborar de alguna forma dentro del sistema, y comunicarse por algún medio con los otros componentes.

Aportando a la definición de arquitectura distribuida, los autores Coulouris, Dollimore, Kindberg y Blair (2012) adicionan otro punto de vista al mismo y mencionan que son componentes de hardware y software localizados y comunicados en red que coordinan sus acciones mediante transmisión de mensajes. Un aspecto para resaltar en esta definición es que dichos componentes se encuentran en red para que su comunicación sea totalmente efectiva. En la actualidad existe una gran variedad de arquitecturas distribuidas, por ejemplo, arquitecturas multiprocesador, arquitectura cliente-servidor, arquitectura de objetos distribuidos, arquitectura punto a punto, entre otras, pero para esta investigación, solamente es necesario entender a qué se refiere el concepto de arquitectura distribuida.

Por lo tanto, se define que una arquitectura distribuida, es un conjunto de componentes de hardware y software totalmente autónomos, que se encuentran comunicados en red con el objetivo de repartir la funcionalidad y balancear la carga de trabajo sobre cada uno de los componentes que forman parte de un sistema.

Para continuar completando los conocimientos necesarios para llevar a cabo este caso de estudio, se requiere comprender dos tipos de arquitecturas distribuidas: la arquitectura orientada a servicios, o mejor conocida como SOA por sus siglas en inglés, y la arquitectura de microservicios. Ambas son fundamentales y ayudarán a plantear la propuesta arquitectónica de este caso de estudio.

- **Arquitectura orientada a servicios (SOA)**

Si bien las arquitecturas distribuidas han ayudado a implementar soluciones empresariales robustas y escalables, en este caso se profundizará un concepto que viene a evolucionar este tipo de arquitecturas. Según Rosen, Lublinsky y Smith (2008), la arquitectura orienta a servicios se define como un estilo arquitectónico que promueve el concepto de servicios empresariales alineados con los del negocio como la unidad fundamental del diseño, construcción y composición de soluciones empresariales.

Para Krafzig, Banke y Slama (2004) este tipo de arquitectura aumenta la capacidad de las empresas para hacer frente a los nuevos requerimientos de negocio a corto plazo, reutilizando la lógica empresarial y los modelos de datos existentes, incurriendo sólo en gastos mínimos de costos, recursos y tiempo, minimizando riesgos, especialmente cuando se reescribe toda la aplicación. Una buena arquitectura orientada a servicios debe proporcionar beneficios duraderos en términos de agilidad, y proporcionar una estrategia a largo plazo para el aumento de la flexibilidad de una infraestructura tecnológica.

Por lo tanto, se definirá que una arquitectura orientada a servicios es un estilo arquitectónico que provee servicios distribuidos, que contienen la lógica empresarial, y cuya finalidad es responder a las exigencias de un negocio cambiante, y proveer una flexibilidad y agilidad para dar respuesta a corto plazo.

Ahora bien, si la arquitectura orientada a servicios brinda la flexibilidad y agilidad necesaria para poder escalar un sistema de software, es necesario abarcar un concepto de servicios que nos ayuda a agilizar y desacoplar aún más la lógica de negocio, que para la presente investigación es un proceso fundamental.

## 2.4 Microservicios

En la actualidad, microservicios es un término muy discutido aproximadamente desde el año 2013, y este es uno de los conceptos más importantes dentro de esta investigación, puesto que es la base del diseño arquitectónico que se propondrá como solución.

Según Flowers y Lewis (2014). Los microservicios son una aproximación de desarrollo que consiste en construir una aplicación sencilla como un conjunto de servicios pequeños, es

decir, que una aplicación construida con microservicios es una aplicación que sigue el estilo de una arquitectura orientada a servicios.

Según Newman (2015), define microservicios como servicios pequeños y autónomos que trabajan juntamente. La pregunta que surge bajo esta definición es ¿cuál es el significado de pequeño? ¿Será que se refiere a un servicio con cuatro líneas de código?

Para Newman (2015), un servicio pequeño es un servicio independiente, que cumple con una responsabilidad única, es decir, con una única funcionalidad.

Los autores Nadareishvili, Mitra, McLarty y Amundsen (2016) en la página 6, definen microservicios desde un punto arquitectónico: "microservicio es un componente desplegable independiente de alcance limitado que soporta interoperabilidad a través de una comunicación basado en mensajes. Una arquitectura de microservicios es un estilo de ingeniería altamente automatizado, un sistema evolucionable compuesto por capacidades alineadas a microservicios".

Es muy importante tomar en cuenta que a aparte que son componentes independientes y con funcionalidad única, deben soportar interoperabilidad, es decir, independientemente el sistema operativo, el lenguaje de programación que se utilizó para desarrollar los microservicios, o el protocolo de comunicación a utilizar, éste lo debe soportar para que este componente independiente cumpla también con el criterio de portabilidad.

Microservicios también se puede definir entendiendo su propósito: "El propósito de microservicios es dividir un sistema de información grande en pequeñas partes, en el cual cada uno posee su propio almacén de datos.". Wolff (2016), página 6. Otro dato que sobresale de esta definición es que sirven para dividir en pequeñas funcionalidades el núcleo de una aplicación, tomando en cuenta que cada una de ellas posee su respectivo almacén de persistencia.

Entre las ventajas más representativas de los microservicios, podemos mencionar:

- Los microservicios pueden distribuirse en varios contenedores de aplicaciones que pueden estar en varias redes informáticas. Richards (2016) expone que los microservicios son arquitecturas basada en redes informáticas y estos pueden almacenarse en los dispositivos de la red he interactúan entre sí mediante una gran cantidad de protocolos de comunicación

como SOAP, JSM, entre otros, lo cual nos lleva a considerar la siguiente ventaja que se interrelaciona con los que expresa Newman (2015).

- Los microservicios no dependen de una tecnología en particular, es más, en ocasiones puede darse el caso de que se mezclen tecnologías al mezclar varios microservicios que las implementen. Newman (2015) llama a esto heterogeneidad de tecnología
- Los microservicios pueden agruparse por áreas de negocio y pueden ser administrados por diferentes equipos. Newman (2015) menciona que los microservicios pueden agruparse en módulos que pueden representar áreas diferentes del negocio y pueden interactuar entre, cada módulo estaría formado por servicios con un alto nivel de cohesión debido que cada servicio obedece a la misma área de negocio.
- Los microservicios pueden ser componentes reusables, de hecho, por definición se espera que sean componentes reusables y aislados que puedan ser mantenidos independientes de cualquier otro sistema Newman (2015).

Por lo tanto, después de contrastar estas definiciones sobre tres puntos de vistas diferentes, se entenderá el concepto de microservicios como componentes desplegables totalmente independientes, autónomos y con bajo nivel de acoplamiento de un sistema de software, que poseen responsabilidad única con el propósito de soportar la interoperabilidad, escalabilidad y mantenibilidad de software en cada uno de ellos.

Cada una de estas características que soportan una arquitectura de microservicios, se definirán a continuación, con el propósito de entender los beneficios que brinda implementarla en el entorno que lo permita.

- **Interoperabilidad**

La interoperabilidad es una característica que provee la arquitectura de software distribuida mediante microservicios, y es definida por el autor Bass, Clements y Kazman (2012), página 121 de la siguiente manera: “Interoperabilidad se refiere al grado en que dos o más sistemas pueden intercambiar información útil a través de interfaces en un contexto particular.”. Este concepto es fundamental para los microservicios, debido a que, si cada módulo desplegable es autónomo, independiente y no atado a una tecnología en específico, éste debe ser capaz de

comunicarse con el resto de los módulos sin importar la tecnología en que ellos fueron definidos, y no generar ninguna complicación al momento de implementar alguna vía de comunicación entre ellos.

La definición del autor Bass, Clements y Kazman (2012) se considera que es muy completa, concisa y clara; por lo tanto, la definición que brinda el autor con respecto a Interoperabilidad es la que se entenderá y se manejará durante el desarrollo de la presente investigación.

- **Escalabilidad**

Según el autor Oquendo, Leite y Batista(2016), página 143, la definición de escalabilidad es el siguiente: "Es el atributo de calidad que se refiere al grado en que un sistema puede ser escalado eficientemente para aumentar el uso operacional.", y es que hoy en día, las necesidades del negocio en las empresas van cambiando a pasos grandes por temas de competencia, ir a la vanguardia, y diferentes razones, y esto puede llegar a requerir que el sistema sea escalado tanto horizontal como verticalmente, dependiendo de la necesidad de la empresa y su entorno. Si la empresa llegara a tener un aumento de clientes y por ello su demanda de servicios crece exponencialmente, es muy probable que su sistema se vea en la obligación de ser escalado para soportar la alta transaccionalidad que éste presente.

Coulouris, Dollimore, Kindberg y Blair (2012), página 19 profundiza en la definición de escalabilidad de la siguiente manera: "Un sistema es descrito como escalable si este se mantiene eficiente al haber un incremento significativo en el número de recursos, funcionalidades y usuarios de este". Es decir que un sistema escalable se define como tal si puede mantener su eficacia al crecer exponencialmente a raíz de modificaciones hechas en el tiempo y/o la carga de nuevos usuarios a través del tiempo

A partir de dichos conceptos, dentro de esta investigación se comprenderá el concepto de escalabilidad como un atributo de calidad dentro de una arquitectura distribuida mediante microservicios, que permite que un sistema sea escalado, ya sea horizontal o verticalmente, con el propósito de soportar un alto uso operacional, y su crecimiento a nivel de componentes y funcionalidad de este.

- **Mantenibilidad**

Esta es una característica muy importante para tomar en cuenta al momento de diseñar una arquitectura de un sistema, con el objetivo de complicar o facilitar a futuro el mantenimiento de este, ya sea para agregar o quitar funcionalidad, o más bien, rediseñar un módulo.

Según el autor Bass, Clements y Kazman (2012), página 10 el concepto de mantenibilidad se puede definir de la siguiente manera: “Es el grado de efectividad y eficiencia con que un producto o sistema puede ser modificado por los encargados de este.”. Es muy importante la definición del autor cuando indica el grado de efectividad y eficiencia que debe implicar, puesto que el software de alguna u otra manera se puede modificar para que soporte una nueva funcionalidad o no, pero el tema acá no es si se puede llevar a cabo o no dicho cambio, sino más bien, en términos de productividad y tiempo invertido para realizar el cambio en el sistema.

En este estudio se entenderá mantenibilidad como un atributo de calidad, que indica el grado de eficiencia y efectividad en que un sistema tiene para ser modificado, a menor tiempo invertido y a mayor productividad para realizarlo.

## 2.5 BPM/BPMS

Es bien sabido que el objetivo de las empresas es generar valor mediante la creación de productos o servicios que satisfagan las necesidades de sus clientes y en un mundo tan competitivo como el actual las empresas deben de buscar la forma de generar un valor agregado a sus productos o servicios para poder competir en un mercado tan cambiante, esto se puede lograr de 2 formas, mediante la innovación de productos o servicios o mediante la optimización y adaptabilidad de los procesos de negocio para mejorar la adaptabilidad y competitividad de la empresa y esto último es precisamente lo que busca BPM, mejorar la gestión de los procesos de una empresa para mejorar su adaptabilidad al mercado como señala S.A. White (2009).

El business process management conocido como BPM es como su nombre lo indica un proceso de gestión de procesos de negocio que tiene la particularidad de ser adaptable y estar enfocado a manejar o gestionar la interacción entre los procesos de negocio, según el Libro BPM (2011), página 6 “BPM es un sistema de gestión enfocado a perseguir la mejora continua del

funcionamiento de las actividades empresariales mediante la identificación y selección de procesos y la descripción, documentación y mejora de los mismos, partiendo del despliegue de la estrategia de la organización, asegurando  a misión empresarial y alineada a la visión de la empresa".

Para R. Laurentiis (2010) la piedra angular para aclarar que es BPM es entender el significado de procesos dentro de las empresas, ya que es una técnica de gestión de procesos de negocio, podemos decir que orienta la gestión de las empresas mediante sus 3 dimensiones principales: el negocio, el proceso y la gestión. Como describe K.Garimella, M. Lees y B. Williams (2008) podemos decir que el negocio es la dimensión del  que crea valor para las empresas orientado a satisfacer las necesidades de sus clientes, el procesos es el medio por el cual se transforman los recursos de la empresa en productos para sus clientes, es decir es la dimensión de transformación que es lo que señala Laurentiis (2010) como la parte fundamental sobre la cual opera BPM y la dimensión de capacitación o de gestión que es un proceso de mejora continua mediante el cual se organizan los procesos que generan valor para la empresa así como los que la hacen funcionar.

Contrario a lo que algunas veces se cree, BPM no es un término que competa exclusivamente a la informática, como hemos descrito está más enfocado a gestionar los procesos de una empresa, pero estos procesos pueden mejorarse mediante la implementación de herramientas de software, pero como lo señala K.Garimella, M. Lees y B. Williams (2008), estas no son más que simples piezas en el proceso de generación de valor y es un error muy común creer que al automatizar parcial o totalmente ciertos procesos de una empresa podemos mejorar la gestión de procesos dentro de la empresa, partiendo de esta premisa podemos hablar de los BPMS o sistemas de gestión de procesos que son herramientas de software que buscan agilizar la gestión de procesos de la empresa, pero estas herramientas BPMS no son herramientas de software mágicas que automáticamente nos ayudan a gestionar los procesos de las empresas, es por eso que S.A White(2009) señala la importancia de poder hacer un modelado detallado y minimalista de los subprocesos de la empresa para poder adaptar estos procesos a que sean manipulados por un sistema de este tipo.

Por lo expuesto podemos decir que los BPMS son sistemas de gestión de procesos empresariales que tienen la posibilidad de brindar gran adaptabilidad al cambio a los procesos de

negocio de una empresa, pero para poder implementarlos es necesaria la existencia de una arquitectura preferiblemente de microservicios.

## 2.6 Factibilidad

Antes de comenzar con un proyecto no sólo de desarrollo de software sino en general, es conveniente hacer al menos un estudio de factibilidad, como lo señala PMI (2013) en PMBook 5ta edición, todas las organizaciones tienen diferentes nociones de cómo llevar a cabo un estudio de factibilidad y todas los trataran de una manera deferente, para unas empresas puede ser un proyecto aparte, otras pueden verlo como un anteproyecto del proyecto que se desea desarrollar o puede incluirse como una fase en el ciclo de vida de desarrollo de un proyecto, pero independientemente como se considere este estudio es innegable de la importancia de llevar a cabo un estudio de factibilidad antes de empezar un proyecto.

Para G Munguía, E. López (2012) el estudio de factibilidad lo incluyen como una buena práctica en la realización de un proyecto, pero más que una buena práctica debe ser considerada como una parte indispensable en el desarrollo del proyecto, G Munguía, E. López (2012) en la página 28, también define la factibilidad de gestión es decir que: “Los tiempos estimados para la realización del proyecto son idóneos y los recursos asignados están disponibles para afrontar el esfuerzo estimado, como lo señalamos anteriormente la factibilidad puede tratarse de diferente manera al inicio de un proyecto”.

Seres (2001) divide la factibilidad en viabilidad económica, técnica y social mientras que Meza (2015) habla de factibilidad económica, técnica y de operaciones de donde podemos concluir que un estudio de factibilidad contempla 3 dimensiones diferentes, la factibilidad técnica, la económica y la operativa. Por tanto, definiremos la factibilidad como la una de las fases iniciales en el desarrollo de un proyecto que pretende determinar la viabilidad de desarrollar un proyecto viéndolo desde el punto de vista económico, operativo y técnico, determinando si los recursos necesarios en cada una de estas áreas para llevar a cabo el proyecto están disponibles o pueden estarlo.

También podemos decir que la factibilidad operativa está determinada por la disponibilidad de todos los recursos necesarios para llevar adelante un proyecto, esto es lo que. Seres (2001) señala

como factibilidad social, aunque esta es sólo una parte de lo que engloba la factibilidad operativa.

La factibilidad técnica está relacionada con encontrar las herramientas, los conocimientos, las habilidades y las experiencias necesarias y suficientes, para hacer que el proyecto sea exitosamente realizado.

La factibilidad económica surge de analizar si los recursos económicos y financieros necesarios para desarrollar las actividades pueden ser cubiertos con el capital del que se dispone o pueden ser adquiridos antes o durante la realización del proyecto.

## 2.7 Mapa conceptual

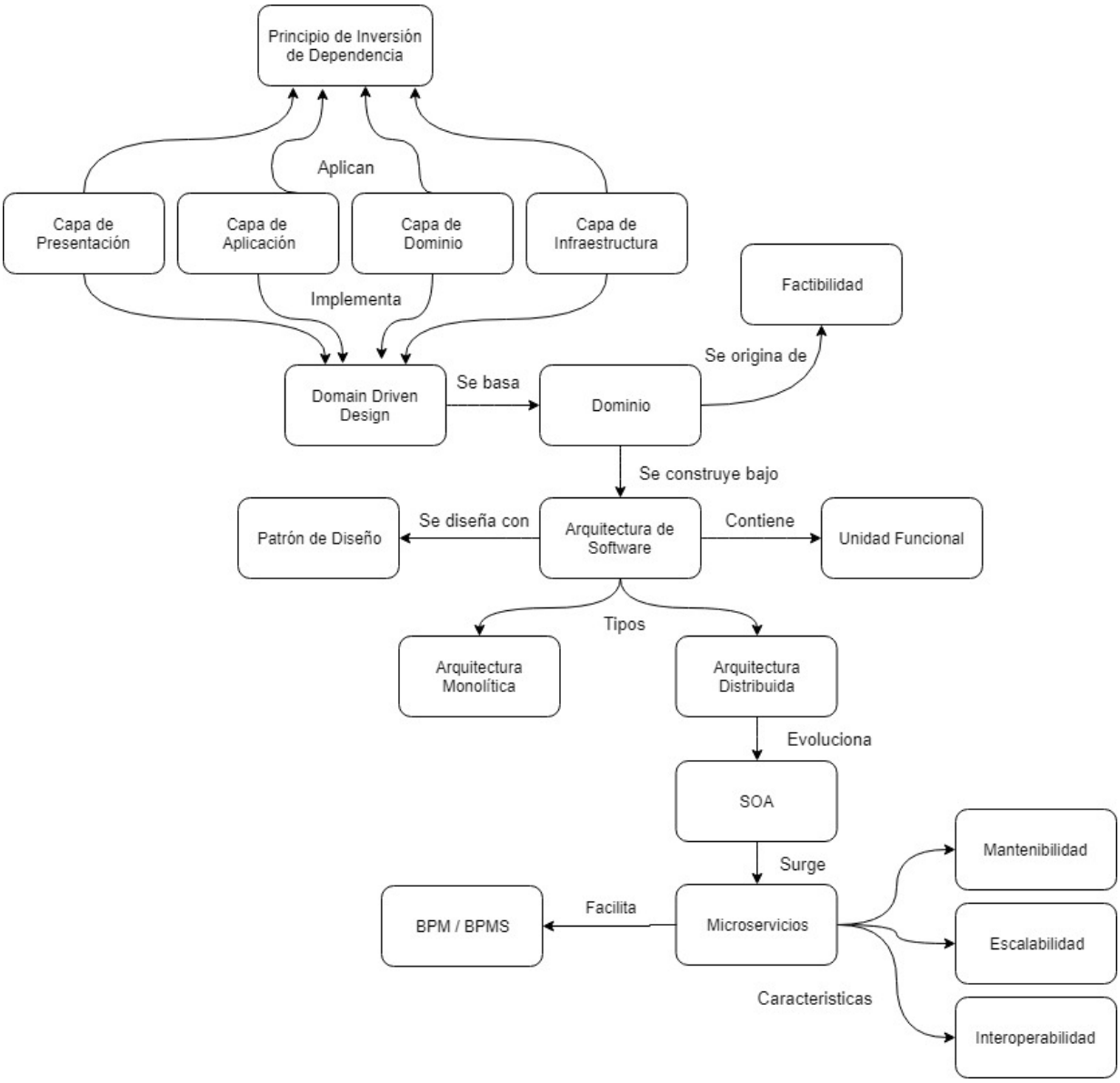


Imagen 1. Mapa conceptual de investigación. Elaboración propia.

# Capítulo 3: Marco metodológico

## 3.1 Tipo de investigación

Según Rivero (2008), una investigación cualitativa descriptiva se utiliza para recoger valoraciones subjetivas de la realidad investigada, por lo que los resultados de la investigación son apreciaciones conceptuales de ideas o conceptos que reflejan de manera fiable y precisa la realidad investigada mediante la caracterización del caso de estudio, señalando sus características y propiedades, en este caso el proceso de descomposición de dominio utilizando un análisis DDD con el objetivo de crear un sistema utilizando una arquitectura de microservicios. Taylor y Bogdan (1994), establecen que los instrumentos descriptivos para recabar información en esta investigación, deben de estar orientados a recolectar las valoraciones de las personas involucradas o conocedoras del caso de estudio concreto o similar, por lo que estos están orientados a recolectar las experiencias, opiniones y valoraciones de los expertos consultados en proyectos de análisis DDD o proyectos de diseño y desarrollo de aplicaciones utilizando una arquitectura de microservicios.

La investigación también es de tipo propositiva por cuanto parte de la necesidad de la empresa de conocer una metodología para poder diseñar una aplicación utilizando una arquitectura de microservicios, partiendo de un análisis DDD de descomposición de dominio.

El tipo de investigación definida para este caso de estudio es mixto y se clasifica como descriptiva-propositiva.

## 3.2 Definición de la metodología de la investigación

Para describir la metodología de la investigación realizada para este caso de estudio, se hará por cada objetivo específico:

1. **Identificar los componentes del dominio que se migrarán a una arquitectura de microservicios.**
    a. Revisión documental de la aplicación: en esta fase de la investigación se buscó comprender el proceso de diseño y construcción de la aplicación en cada una de

sus capas (capa de presentación, capa de dominio, capa de aplicación), de esta forma identificar sus componentes, sus procesos, la infraestructura donde se aloja, su rendimiento, promedio de transacciones atendidas, casos de uso, requerimientos funcionales y no funcionales, entre otros. Esto con el fin de comprender el contexto de la aplicación.

b. Revisión documental del proceso de modelado de dominio: esta revisión nos ayudó a comprender como se ha aplicado el principio de inversión de dependencia en el proceso que se ha llevado a cabo para la descomposición del dominio, y la separación de la lógica de negocio de las otras funcionalidades que maneja la aplicación, la cantidad de subdominios en los que se ha separado las unidades funcionales de la aplicación y el criterio utilizado para crear los mismos, esta revisión se complementó con una entrevista descriptiva hechas al personal que participo en la migración, según lo especificado en el apéndice 2.

c. Revisión bibliográfica acerca del proceso de modelado de dominio: En esta etapa se hará una revisión acerca de los pasos a seguir al realizar un análisis de descomposición de dominio utilizando DDD con el objetivo de corroborar el correcto proceso realizado para hacer el análisis de descomposición de dominio y proponer posibles oportunidades de mejora en el mismo.

d. Validar con expertos la factibilidad del modelo de dominio: En esta etapa se validó que el modelo de dominio realizado por el departamento de TI es adecuado para aplicar una arquitectura basada en microservicios y de ser necesario, se propondrán recomendaciones para mejorar el modelado del dominio. (Revisar apartado de apéndice 1, para conocer el guion de preguntas de dicha entrevista). Adicionalmente, las personas expertas que fueron elegidas para llevar a cabo esta validación cumplieron con el perfil que se especifica en el apéndice 3 de este trabajo.

**2. Definir una arquitectura distribuida usando microservicios partiendo del modelo de dominio.**

a. Revisión bibliográfica de patrones de diseño: la revisión bibliográfica nos permitió expandir los conocimientos acerca de los patrones de diseño que pueden

utilizarse en la propuesta final, estos patrones deben ir orientados a solventar los problemas de mantenibilidad y escalabilidad de la aplicación. Los patrones de diseño propuestos serán seleccionados de acuerdo con los criterios de evaluación de escalabilidad y mantenibilidad propuestos en las normas ISO 25010/2011, aunque no se descarta la utilización de otras normas de evaluación como las propuestas por W3C.

b. Revisión bibliográfica acerca de microservicios: En esta etapa se hará una investigación bibliográfica de que son y cómo implementar la arquitectura distribuida de microservicios, así como sus debilidades y fortalezas, con el objetivo de servir como base para proponer un diseño arquitectónico utilizando microservicios y sustentar la creación de una guía de buenas prácticas para realizar un análisis DDD aplicado a una arquitectura de microservicios
c. Diseñar la propuesta arquitectónica utilizando microservicios y los patrones de diseño seleccionados. (ver objetivos de apéndice 2)

3. **Elaborar una guía de buenas prácticas para realizar un análisis domain driven design aplicado a microservicios**
    a. Elaboración e implementación de instrumentos descriptivos que permitan recolectar opiniones de personal experto tanto en el área de análisis y modelado DDD como en la implementación de microservicios, estos instrumentos contendrán preguntas abiertas que permitan a los expertos expresar las valoraciones que consideren importante acerca de los temas antes expuestos
    b. Elaboración de una guía de recomendaciones y buenas prácticas para la migración de una arquitectura monolítica a una arquitectura de microservicios utilizando DDD. (ver objetivos de apéndice 2).

La matriz de congruencia se encuentra en la sección apéndices y detalla los instrumentos, indicadores y variables utilizados en la metodología de investigación para este caso de estudio.

### 3.3 Instrumento descriptivo

En el desarrollo de esta investigación, se programaron entrevistas a expertos para conocer su valoración con respecto al caso de estudio que planteamos, con el objetivo de enriquecer y

fundamentar los resultados. Para las entrevistas se planteó un guion que nos ayudó a llevar a cabo cada pregunta con el experto. Estos instrumentos se han desarrollado a partir de las especificaciones de Taylor y Bogdan (1994) acerca de cómo debe de estructurarse una entrevista como instrumento descriptivo.

| **Puntos a abordar** | **Dimensión** | **Objetivo** | **Preguntas** |
|---|---|---|---|
| **Validación de descomposición de dominio** | Contexto | Contextualización a experto | **Input: Modelado de dominio usando DDD** |
| | | | 1. ¿Por qué se busca llevar la aplicación de una arquitectura monolítica a una distribuida? Contextualización con experto |
| | Contenido | Contextualización a experto | 1. Descripción del proceso de descomposición de dominio de la aplicación monolítica |
| | | | 2. Los contextos aislados se han armado a partir de los grupos de subdominios que forman parte de la misma funcionalidad, como comúnmente se hace, ¿pero considera que hay otra manera de agrupar estos subdominios? |
| | Valoración | Obtener información, consideración, recomendaciones de expertos | 1. En base a la contextualización y descripción del proceso de descomposición, validar y recibir retroalimentación del experto al respecto. |
| | | | 2. ¿Bajo qué consideraciones usted cree que se lleve de lo monolítico a lo distribuido? ¿Cuál hubiera sido el camino que usted hubiera tomado? |
| | Conclusión | Fundamentar comentarios del experto | 1. ¿Será necesario descomponer aún más el dominio? ¿por qué? |
| | | | 2. Se tomarán en cuenta todas las consideraciones dentro del proceso de descomposición del dominio, según el experto |

Tabla 1. Instrumento descriptivo para entrevista con expertos

# Capítulo 4: Análisis del caso

## 4.1 Descripción de la compañía

Esta compañía es una empresa de transporte aéreo y fue fundada en 1919. Es la segunda aerolínea más grande de Suramérica después de LATAM. Sus principales centros de conexión se encuentran en el aeropuerto el Dorado de Bogotá, el aeropuerto internacional de El Salvador y el aeropuerto internacional Jorge Chávez de Lima.

La compañía tiene una planilla por más de 20,500 colaboradores y una flota compuesta por más de 200 aeronaves, que operan a más de 100 destinos en 26 países en América y Europa y ofrece un promedio de 5100 vuelos semanales.

Luego de un proceso de evolución de la compañía, en 2010 oficializaron su fusión con otra aerolínea latinoamericana, lo cual dio marcha a un riguroso proceso de reorganización administrativa, así como de integración de sus redes de rutas, homologación de procesos y captura de sinergias. Y actualmente la compañía está constituida por un grupo de 7 aerolíneas subsidiarias.

## 4.2 Descripción general del sistema

Como se ha descrito en el apartado [1.2] el sistema analizado para este caso de estudio fue desarrollado en el año 2011 en respuesta a la fusión de dos compañías del transporte aéreo, el cual tenía como propósito la homologación de los procesos de negocio de dos sistemas diferentes bajo un mismo núcleo.

El término de compensación en lenguaje del negocio: es el medio por el cual se trata de compensar al cliente con el objetivo de retribuir cualquier inconveniente o falla que pueda generarse en el servicio que le brinda la compañía, y de esta forma tratar de conservar su lealtad.

En el día a día es necesario registrar y contabilizar las compensaciones generadas en los puntos de abordaje por diversos motivos, por ejemplo: demoras, cancelaciones, fallas mecánicas de última hora, entre otros. Existen además diversas áreas de negocio que hacen uso de compensaciones, entre ellas están: call center, carga, equipajes, aeropuertos, puntos de ventas.

Para dar respuesta a la necesidad de controlar las compensaciones, el sistema cuenta con un portal web que es utilizado específicamente por el área de puntos de abordaje, y a su vez expone un servicio con toda la lógica para el manejo de una compensación. Este servicio es utilizado por aplicaciones de otras áreas de negocio que hacen uso de la compensación a clientes.

La arquitectura de la solución ha sido diseñada utilizando un patrón en capas, en la que cada capa está constituida por componentes específicos según su función. Las capas diseñadas están constituidas por: la capa de vista, que es la encargada de la presentación a los usuarios, una capa lógica, la cual maneja toda la lógica de la aplicación, y una capa de datos, la cual maneja todas las interfaces tanto internas e interfaces externas a otros sistemas.

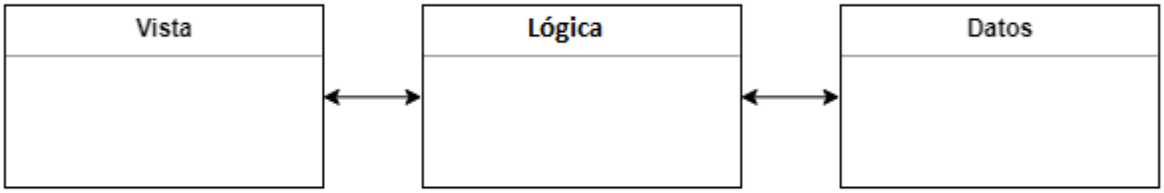


Imagen 2. Arquitectura monolítica en capas. Proporcionado por la compañía.

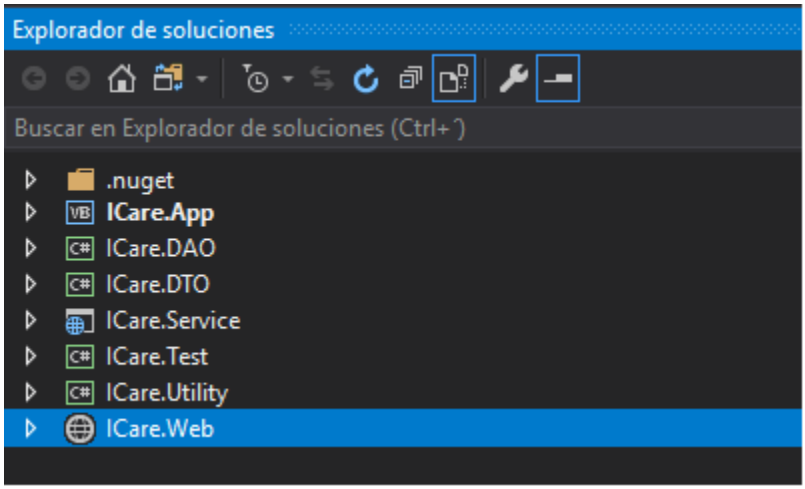


Imagen 3. Vista de la solución del sistema. Proporcionada por la compañía.

En la solución del sistema se puede observar que la capa de vista está representada por el proyecto de ICare.Web, la capa de lógica está representada por el proyecto ICare.App, la capa de datos está representada por los proyectos ICare.DAO y ICare.DTO, el proyecto ICare.Service

representa el servicio web expuesto por el sistema y que comparte la misma lógica y datos que la aplicación web.

A continuación, se presenta la información general del sistema.

| **Campo** | **Descripción** |
|---|---|
| Descripción de la aplicación | Sistema de registro y emisión de compensaciones generados por diferentes canales y áreas de negocio de la compañía. También da soporte al proceso de compensación por irregularidades de vuelos y sobre ventas. |
| Naturaleza | Aplicación web |
| Servidor de aplicación | Dos servidores balanceados. |
| Servicios que expone | Web service para:<br>• Registro de pasajero afectado<br>• Registros de motivos de afectación<br>• Solicitud de emisión de compensaciones<br>• Solicitud de anulación de compensaciones. |
| Segregación | Desarrollo, pruebas y producción. |
| Alta disponibilidad | Si |
| Estándar de desarrollo | • Utilización de programación orientada a objetos para encapsular métodos y funcionalidad utilizada por otros sistemas o componentes.<br>• Control de excepciones<br>• Tres capas |
| Proceso de negocio que soporta | • Emisión de compensaciones por irregularidades de vuelo<br>• Control de política de compensación.<br>• Contabilización de servicios. |

Tabla 2. Información adicional del sistema. Proporcionado por compañía.

El sistema cuenta con múltiples componentes que manejan los procesos de negocio. Para entender el funcionamiento, se mencionarán y explicarán sus escenarios.

Carga de irregularidades: Este paquete permite buscar, crear o seleccionar los vuelos que han sido afectados por una irregularidad de vuelos, su fuente de datos son dos sistemas que

podría decirse que tienen la misma información que es la información de vuelos. Uno posee la información de vuelos diario y el otro posee la información de vuelos históricos.

Administración de políticas de compensación: Este paquete permite aplicar las configuraciones de políticas de compensación para las diferentes rutas, permite registrar los motivos de irregularidades de vuelo y seleccionar las compensaciones para dicha ruta y motivo, este paquete es manejado por el administrador del sistema, y con dichas configuraciones de compensaciones los demás usuarios se rijan bajo estas políticas.

Este paquete interactúa con los sistemas para la carga de rutas (vuelos), estas rutas son las entradas que necesita el sistema para configurar las políticas de compensación.

Administración de irregularidades de vuelos: Permite seleccionar los vuelos que han tenido irregularidades de vuelos, y permite registrar los motivos de irregularidad de vuelo a los vuelos seleccionados.

Asignación tipo de compensación: Permite registrar los tipos de compensación o servicios según la política (configuración de compensaciones) que aplica al pasajero efectuado.

Generación y entrega de servicios en puntos de abordaje: Este escenario permite efectuar las emisiones de vouchers de servicios que el usuario de puntos de abordaje o call center otorgue al pasajero afectado.

Generación y entrega de compensaciones: Este escenario permite efectuar las emisiones/entrega de compensaciones que el usuario de puntos de abordaje o call center otorgue al pasajero afectado. Estos paquetes permiten la generación y otorgación de documentos electrónicos como comprobantes de la compensación a través de la interacción con el sistema de servicios, el sistema de millas y el sistema administrador de vuelos todos por medio de servicios web externos.

Facturación: Este escenario permite contabilizar los servicios que se les han brindado a los pasajeros afectados, permite sumariarlos por proveedor y también permite enviar órdenes de compras de estos servicios.

## 4.3 Componentes arquitectónicos del sistema

Con los componentes identificados, es necesario comprender como es la interacción dentro de la lógica de la aplicación. Para esto, se explicará el diagrama de componentes arquitectónicos.

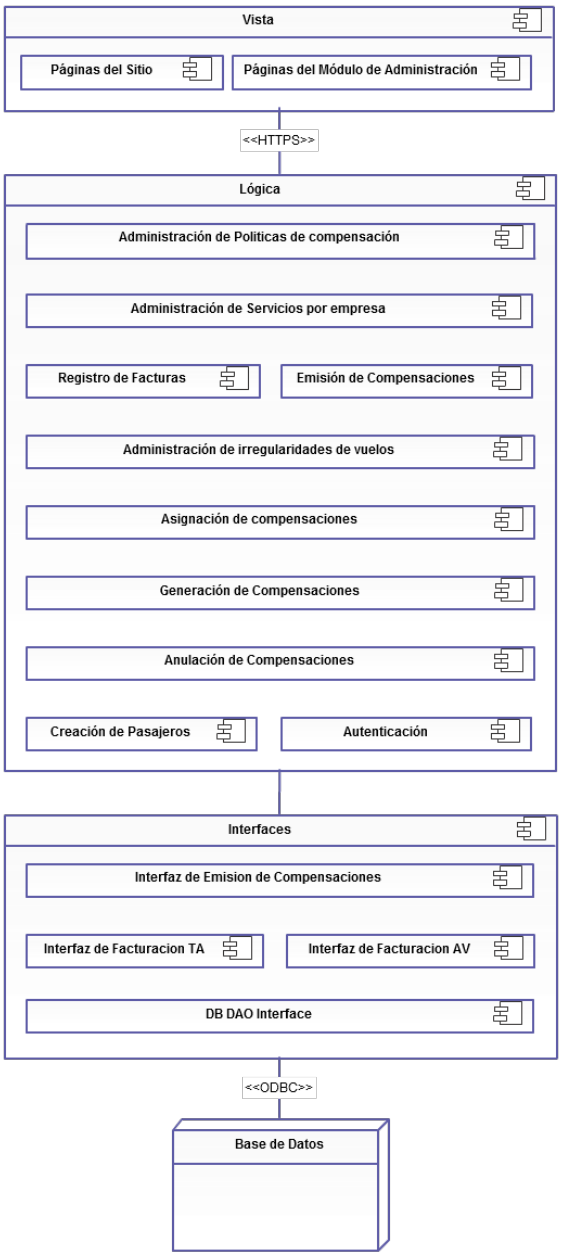


Imagen 4. Componentes arquitectónicos. Proporcionados por compañía.

1. Páginas del sitio: En este componente se encapsulan todas las páginas de la solución.
2. Páginas del módulo de administración: En este componente se encapsulan todas las páginas de la solución que tengan relación con los módulos de administración descritos en los siguientes puntos.
3. Autenticación: Este componente se encarga de la autenticación de los usuarios en la aplicación el cual hará uso del módulo de autenticación de la compañía mediante servicio web de single sign-on.
4. Administración de políticas de compensación: En este componente se realiza la administración de compensación y creación y asignación de políticas por motivo.
5. Administración de servicios por empresa: En este componente se puede realizar la administración de los proveedores y asociar los servicios que pueden brindar las empresas a los pasajeros.
6. Registro de facturas: Se realiza el proceso de validación y registro de las facturas para su posterior procesamiento en el sistema contable.
7. Asignación de compensaciones: En este módulo se determina qué tipo de compensación se le brindará al pasajero afectado según motivos de compensación asignados al vuelo, apegándose a las políticas de compensación.
8. Generación de compensaciones: En este módulo se relaciona la retribución exacta que se le brindará al pasajero afectado en cada uno de los lugares en los que se puede emitir una compensación.
9. Emisión de compensaciones: En este módulo ejecuta la generación de las compensaciones a entregar al pasajero afectado, acreditar millas, efectivo, entre otros.
10. Anulación de compensaciones: En este módulo se realizan las anulaciones de la compensación que tienen un estado de emitido para poder hacer el respectivo rollback según aplique para cada caso.
11. Administración de irregularidades de vuelos: Componente que permite la administración de los vuelos con irregularidades y que son sujetos de compensaciones para los pasajeros que han sido afectados.
12. Creación de pasajeros: Modulo en el cual se administrará la lista de pasajeros afectados por irregularidades en los vuelos.

13. Interfaz de emisión de compensaciones: Interfaz que se expone como servicio para poder registrar las compensaciones emitidas por sistemas de terceros.
14. Interfaz de facturación 1: Componente que permite realizar el procesamiento de los datos de las facturas en el sistema contable utilizado en SAP (sistemas, aplicaciones y productos), con el fin de realizar el pago a proveedores, esta interfaz será expuesta como un servicio web.
15. Interfaz de facturación 2: Componente que permite realizar el procesamiento de los datos de las facturas en el sistema contable utilizado en Oracle Financial, con el fin de realizar el pago a proveedores, esta interfaz será expuesta como un servicio web.
16. Interfaz para base de datos usando DAO (Objeto de acceso a datos, por sus siglas en inglés): Es una interfaz con enlace directo a la base de datos que realiza toda la administración de las cadenas de conexión.

La capa de vista es la encargada de mostrar a los usuarios que se encuentran en los puntos de abordaje, la interfaz gráfica. Esta interfaz interactúa con la capa de lógica, la cual contiene las clases para cada uno de los componentes de negocio. Uno de los principales problemas, es que la capa de lógica contiene un alto acoplamiento, debido a que toda la lógica de negocio está situada en una sola capa. Las interfaces proveen el acceso a la persistencia y a los demás servicios de terceros.

### 4.4 Diagrama de despliegue del sistema

Para entender la necesidad de desacoplar los componentes de negocio del sistema, es importante tener claro el esquema de infraestructura en el cual está desplegado, puesto que no siempre el aumentar los recursos de infraestructura es la solución a problemas de rendimiento y tiempos de respuesta.

Como se ha mencionado en el apartado [1.2], la implementación se hace con una escalabilidad horizontal, es decir, se replica la instancia de la aplicación en dos servidores, por medio de un balanceador de carga. Dicha implementación ayuda a mantener la alta disponibilidad, pero no remedia el problema del rendimiento. También cabe mencionar que los

servidores no son dedicados para esta aplicación, sino que aloja otras aplicaciones de la compañía, por lo que en su entorno siempre hay recursos compartidos.

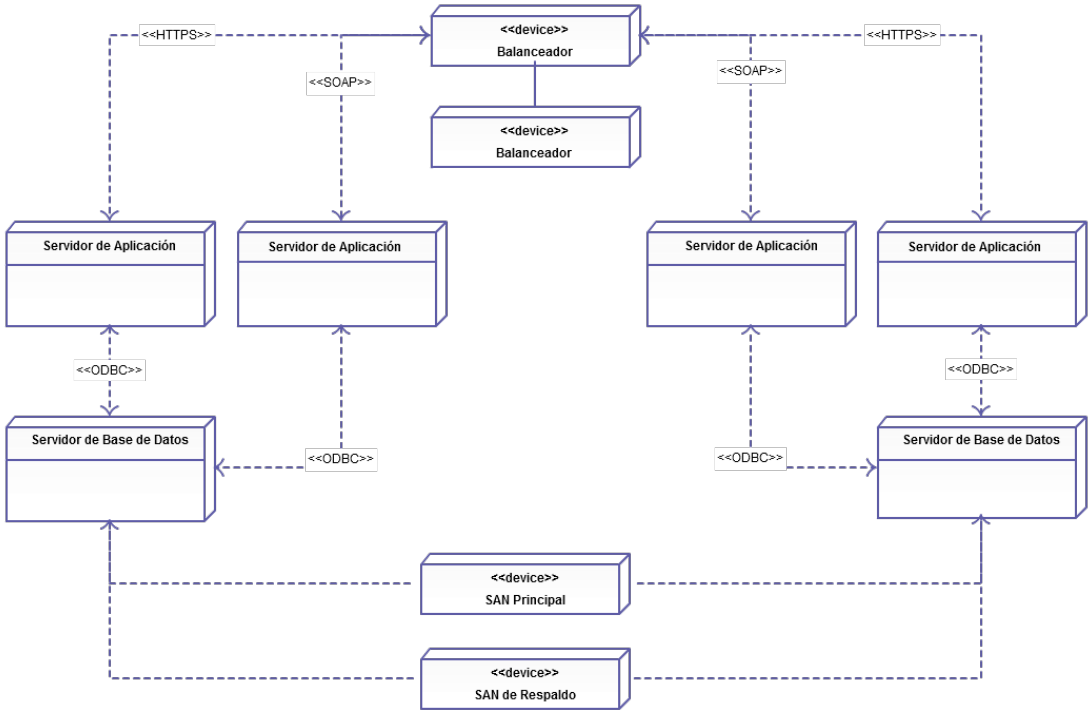


Imagen 5. Arquitectura de despliegue. Proporcionado por compañía.

Donde:

1. Servidor de lógica aplicación: En este servidor se alojan tanto los componentes de lógica de la solución, las páginas aspx de la solución y los componentes que realizarán la interfaz con la base de datos. Los componentes que son dedicados a las interfaces que sean expuestas a terceros deben de encontrarse en un servidor de aplicación diferente al que aloja el sitio web.
2. Servidor de base de datos: En este servidor se almacena la persistencia de la solución.
3. SAN (Red dedicada al almacenamiento, por sus siglas en inglés): Se refiere al espacio compartido entre los servidores para el almacenamiento de la data que permite que estos se encuentren en un esquema fail load en caso de alguna eventualidad sobre alguno.

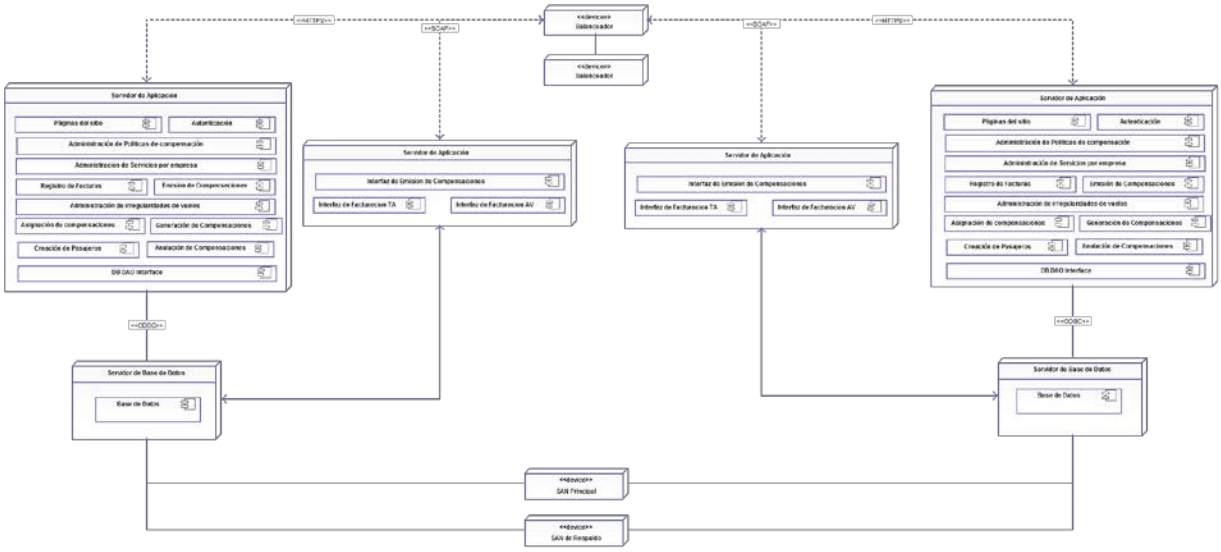

Imagen 6. Componentes de infraestructura. Proporcionado por compañía.

## 4.5 Problemática

Dentro del entorno del sistema, unos de los factores críticos para esta aplicación es el rendimiento y la alta disponibilidad, debido a que la volumetría de las transacciones demanda que sus tiempos de respuesta no afecten la operación en cada punto de abordaje, y por su importancia es necesario que el sistema esté disponible el 99.99% del tiempo.

La problemática concerniente para este trabajo es el rendimiento y la complejidad ciclomática, es decir, el problema para poder comprender y mantener la lógica del sistema.

Si bien, la calidad del software demanda que la documentación técnica sea generada y mantenida durante todo el ciclo de vida, para muchos desarrolladores esta actividad consume mucho tiempo y en muchas ocasiones la documentación existente no está actualizada o simplemente no existe, por lo que la documentación más fácil y rápida para los desarrolladores es comentar el código (Coral Calero, 2010).

La falta de documentación técnica dificulta el entendimiento del código, arquitectura, interfaces, entre otros., y aumenta la complejidad de mantener estándares y orden en el proceso

de desarrollo. Partiendo de este punto, la complejidad de la mantenibilidad del sistema estudiado ha aumentado a un grado que el desarrollo de una nueva funcionalidad puede generar fallas en otra ya existente debido a que no se cuenta con documentación técnica. Por tal motivo, el tiempo de pruebas ha aumentado debido a que es necesario probar todo el sistema para garantizar que no haya afectación.

La falta de estándares y buenas prácticas de desarrollo dentro del departamento de TI y la rotación del personal, ha generado que el código del sistema sea un código espagueti difícil de comprender y escalar, por lo que su complejidad ciclomática ha ido en aumento. Esto ha generado que atender nuevos requerimientos para el sistema se vuelva mucho más costoso en términos de tiempo y es fuerzo y no cumpla con la aceptación del negocio.

Al observar la documentación general del sistema vemos que su arquitectura inicial es en capas, y al profundizar en el código de la solución nos damos cuenta que la capa de lógica tiene mucho acoplamiento para todas las clases que contienen la funcionalidad del sistema, adicionalmente la falta del uso de patrones de diseño, inyección de dependencia, principios SOLID, entre otras técnicas de desarrollo, han hecho que porciones de código estén duplicados en distintas partes del sistema aumentando la probabilidad de fallas cuando se hace una modificación.

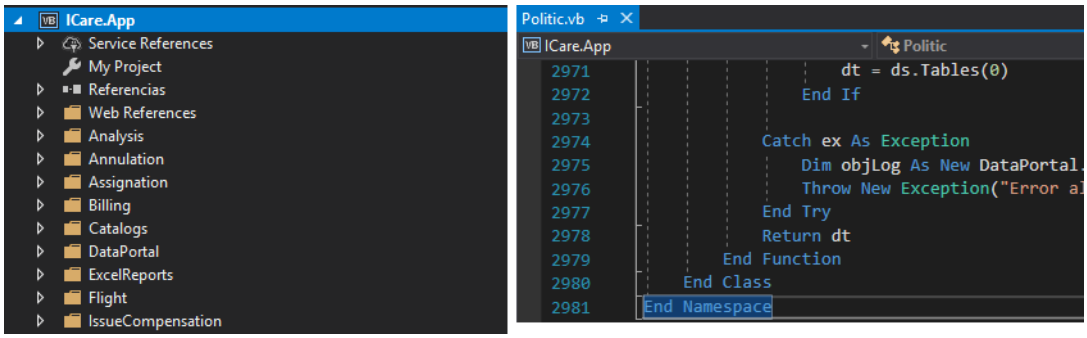


Imagen 7. Vista del proyecto ICare.App y clase Politic.vb. Proporcionada por la compañía.

La capa de lógica contiene diversas carpetas, y cada una contiene las clases para cada funcionalidad. Existen clases que cuentan con más de 2,000 líneas de código lo cual dificulta la compresión de estas.

Una de las métricas utilizada por el equipo de calidad y operaciones es el uso de los recursos del servidor, y este aspecto el problema de la aplicación es que en horas pico puede utilizar el 85% del CPU y recordemos que su infraestructura es compartida con otros sistemas que de igual forma hacen uso de los mismos recursos.

La principal razón del alto uso de recursos en horas pico, es que al ser un sistema monolito se compila y despliega como un único componente, tanto las vistas, el code behind y el servicio web se publican en un directorio virtual ubicado en el mismo servidor, lo cual hace que todo el procesamiento de datos utilice el mismo recurso. Aunque en su arquitectura inicial se definió utilizar dos instancias de la aplicación en dos servidores diferentes administrados por un balanceador de carga para asegurar el rendimiento y la disponibilidad del sistema, esta implementación no radica la complejidad de la aplicación y el uso del recurso.

Actualmente aun teniendo ambas instancias el sistema presentan problemas de rendimiento y se refleja en los tiempos de respuesta a las peticiones. Se ha medido en horas pico que el proceso para compensar un vuelo puede tomar alrededor de 20 a 30 minutos, a esto se le agrega que el sistema es usado en más de 20 países en simultaneo. El pensar que los agentes del punto de abordaje tienen que esperar esa cantidad de tiempo con un grupo aproximado de 130 pasajeros furiosos por perder su vuelo, se vuelve una pesadilla.

### 4.6 Aplicando domain driven design para el nuevo dominio.

Una de las estrategias utilizadas para llevar a cabo una transformación de este tipo, es poder desarrollarla de forma incremental e iterativa. Por esta razón el principal punto de partida que realiza el equipo de TI es el de definir el equipo de trabajo, el cual para este caso ha sido conformado por: desarrolladores, líder técnico, arquitecto de software y expertos del negocio.

Por medio de las reuniones del equipo de trabajo se determinan los componentes del sistema que soportan los procesos de mayor demanda por parte de la operación: emisión de compensaciones, generación de compensaciones, anulación de compensaciones, asignación de compensaciones.

El concepto de DDD definido por Evans (2003) provee herramientas útiles para el modelado de dominio, y es por esto por lo que el equipo de trabajo ha buscado definir un lenguaje entendible tanto para los expertos del negocio como para el equipo de desarrollo, cuyo alcance considera los siguientes aspectos: nombres de las clases y sus métodos, objetos, nombres de patrones de diseño aplicados al modelo de dominio.

Siguiendo la arquitectura de DDD se crean las siguientes capas dentro de la solución: presentación, aplicación, dominio, infraestructura, donde esta última cuenta con la responsabilidad del acceso a datos y la comunicación a los servicios que consume el sistema.

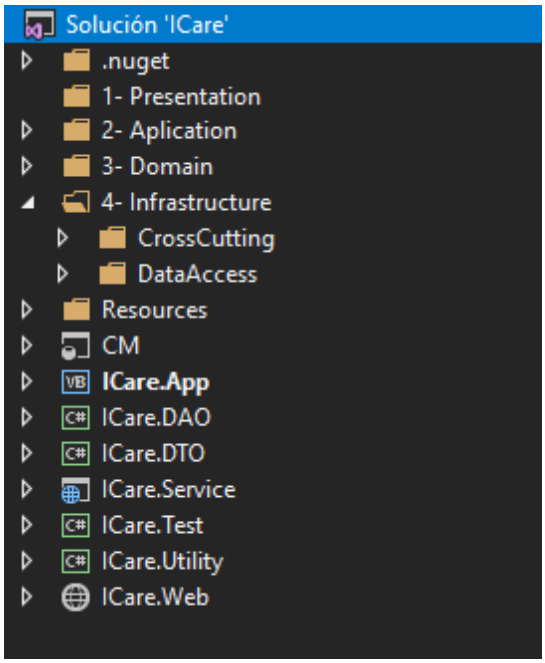


Imagen 8. Vista del proyecto con nuevas capas según DDD. Proporcionada por la compañía.

Tomando en cuenta el tiempo de desarrollo para la transformación de este sistema, el equipo de trabajo opto por reutilizar la parte visual, por lo que sólo se adaptará para que su función sea únicamente la de mostrar y consultar información hacia la capa de aplicación.

Para este caso de estudio la descomposición del dominio se realiza a partir de la lógica de negocio que maneja el sistema, es decir que los módulos de administración y todo el CRUD se mantendrá en la lógica y arquitectura actual.

Dentro de la definición de esta nueva arquitectura basada en DDD únicamente las capas de aplicación e infraestructura interactúan con el dominio. La capa de dominio no posee ninguna

dependencia con otra capa debido a que es portable, genérica y reutilizable independientemente de la tecnología, plataforma o arquitectura que se utilice tal como lo muestra la imagen 9.

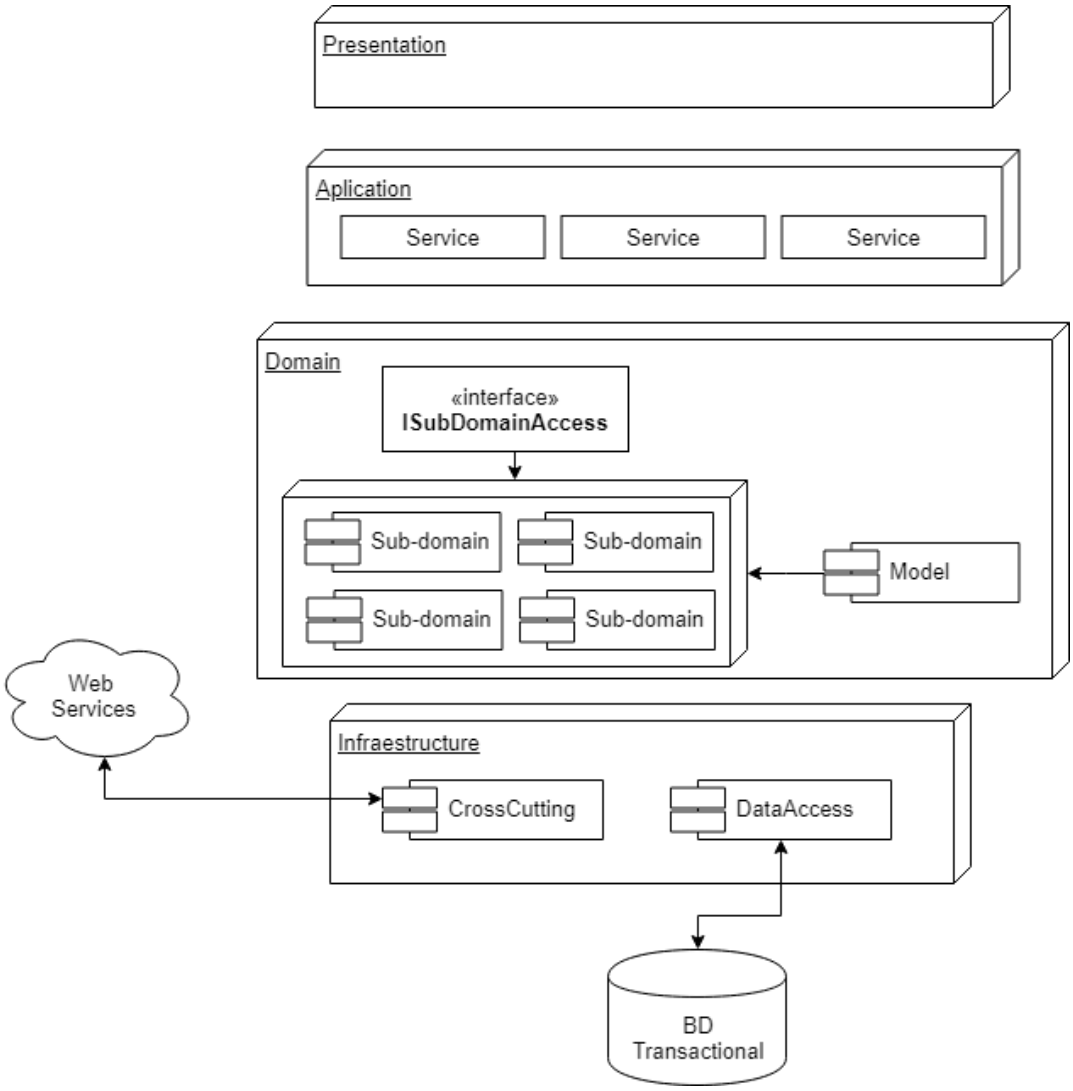


Imagen 9. Vista nueva arquitectura DDD para el sistema. Proporcionada por la compañía.

La capa de aplicación se encarga de orquestar el trabajo que debe hacer el sistema, en esta no se especifica ninguna regla de negocio y no tiene conocimientos del dominio. Por tal motivo, al hacer la revisión de todas las funcionalidades del sistema se definen los siguientes componentes de la capa de aplicación: anulación, cargaProveedores, cargaVuelo, compensaciones, contabilización.

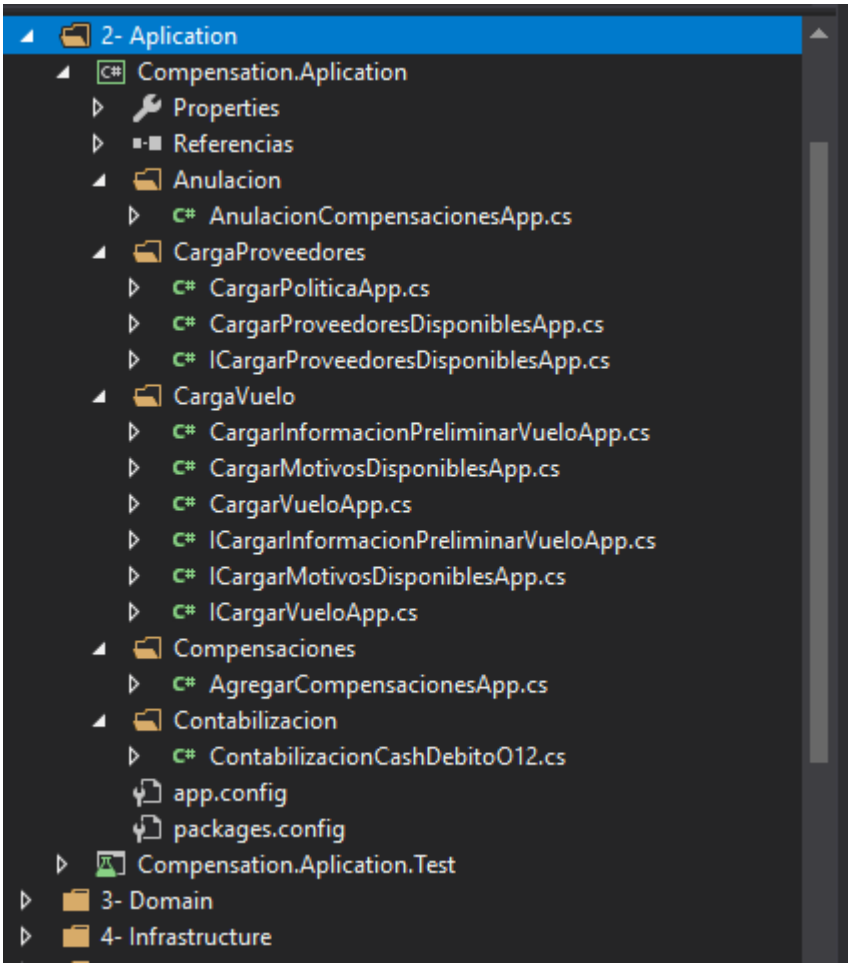


Imagen 10. Vista capa aplicación del sistema. Proporcionada por la compañía.

Cada componente de la capa de aplicación tendrá comunicación con las vistas correspondientes a esas funcionalidades, y al mismo tiempo será el orquestador hacia la capa de dominio. Esta comunicación entre aplicación y dominio se hace por medio de interfaces, para evitar la dependencia entre capas.

En el proceso de modelado es necesario tener en claro cuál es el dominio principal del sistema, y para este sistema el equipo de trabajo ha determinado que el dominio es la compensación. Quiere decir que la razón de ser de la aplicación es compensar a los clientes, aun si intervienen otros actores como proveedores, políticas, rutas., el propósito del sistema es dar una retribución por la interrupción del servicio por el cual el cliente pago.

Una vez identificado el dominio, es necesario descomponerlo en pequeñas unidades de negocio y de responsabilidad única conocidas como subdominios. Recordemos que en esta

descomposición se busca aislar las complejidades del negocio, por lo tanto, de este proceso lo importante es identificar las entidades del modelo, las interfaces por la cual la capa de infraestructura podrá interactuar y los servicios que tienen la lógica y reglas de negocio.

En el proceso de descomposición de dominio se determina el modelo del dominio, en el cual se encuentran las entidades que manejará la capa de dominio, para este caso las entidades se muestran en la siguiente imagen.

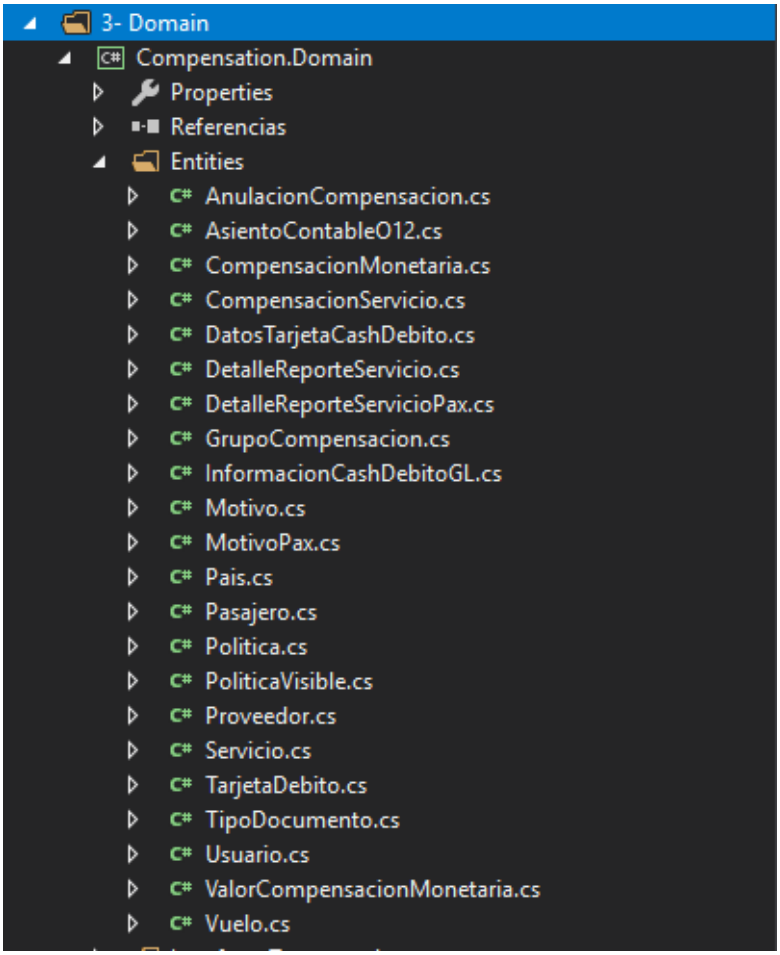


Imagen 11. Vista entidades capa de dominio. Proporcionada por la compañía

Cada entidad se compone por sus atributos y a su vez son utilizados por los servicios de la capa de dominio. Al usar la palabra servicios no hacemos referencia a un servicio web como tal, sino a la funcionalidad de negocio que contiene y expone la capa de dominio hacia la capa de aplicación e infraestructura, pero como tal son clases accedidas por medio de las interfaces.

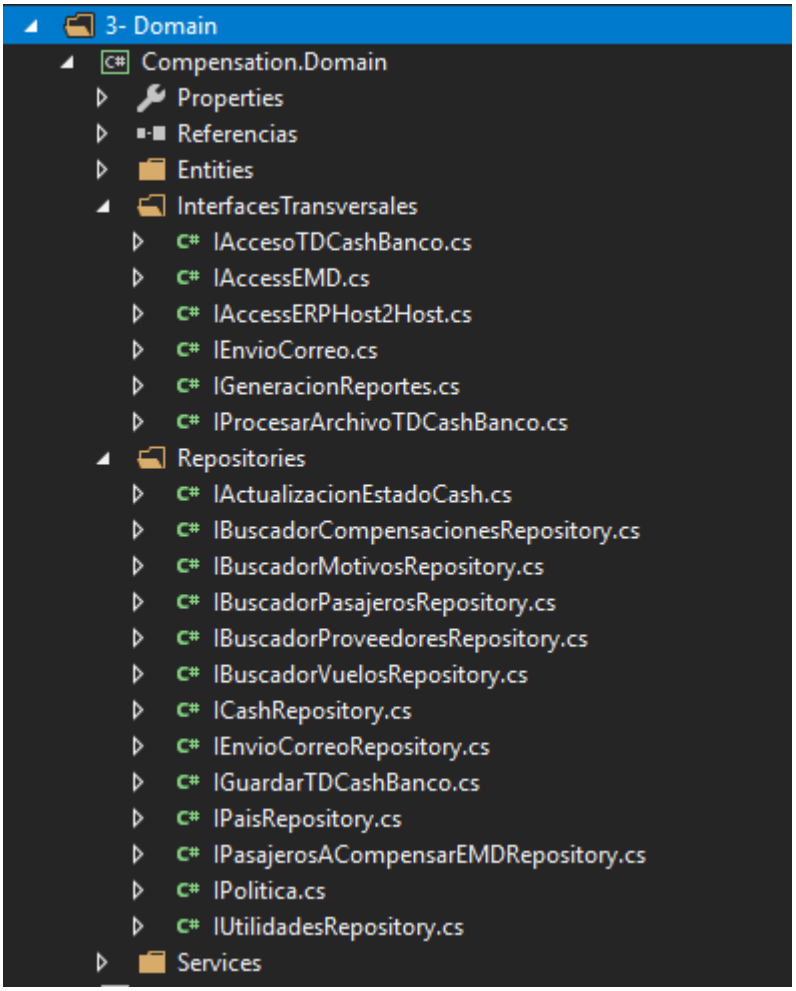


Imagen 12. Vista interfaces capa de dominio. Proporcionada por la compañía

Las interfaces transversales son las interfaces que ocupa el componente cross-cutting de la capa de infraestructura, las interfaces de repositorios son las interfaces que utiliza el componente data access. La capa de aplicación puede tener comunicación con la capa de Infraestructura por medio de las interfaces que implementa, tal como lo sugiere el principio de inversión de dependencias. La orquestación de la capa de aplicación hacia la capa de dominio se ha desarrollado de forma asíncrona, con la finalidad de mejorar el rendimiento y el tiempo de respuesta del sistema.

En la imagen 13 podemos observar los subdominios identificados por el equipo de trabajo, los cuales manejan la funcionalidad del negocio y su lógica no depende de otro. Cabe mencionar que estos componentes tienen responsabilidad única, es decir son responsables de una sola funcionalidad o regla de negocio.

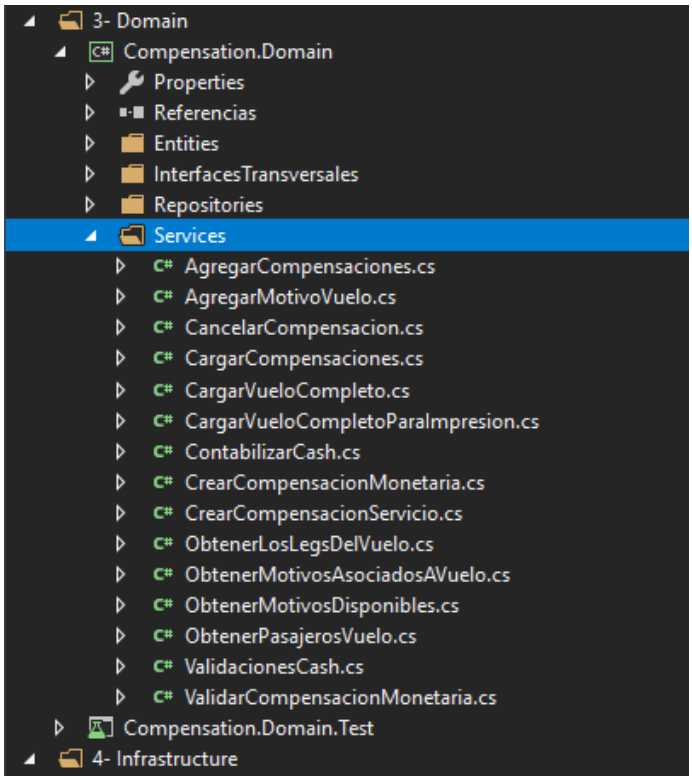


Imagen 13. Vista subdominios capa de dominio. Proporcionada por la compañía

La capa de Infraestructura tiene los componentes cuyo propósito es la extracción y el manejo de la información transaccional, así como de los servicios web que consume el sistema.

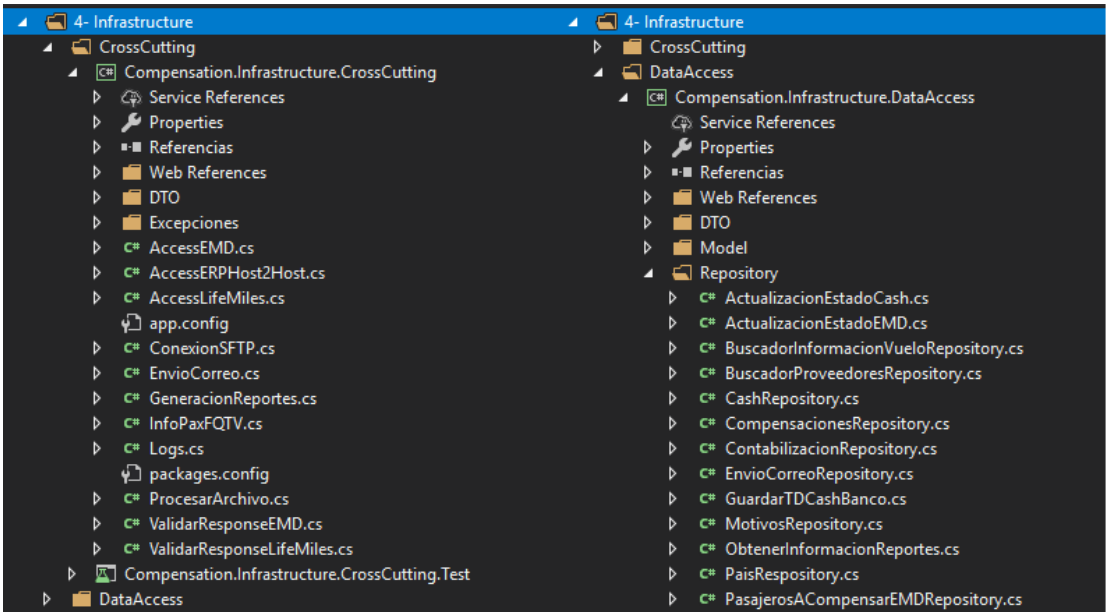


Imagen 14. Vista capa de infraestructura. Proporcionada por la compañía

# Capítulo 5: Propuesta arquitectónica usando microservicios

## 5.1 Arquitectura propuesta.

Con el desacoplamiento de la lógica de negocio como subdominios en la capa dominio de la solución del sistema, se ha abarcado gran parte de la transformación del sistema. Esto debido a que DDD provee herramientas para el modelado de dominios complejos, lo que garantiza que cada subdominio es de responsabilidad única y que no dependen de otro, por lo tanto, hemos reducido el acoplamiento de la funcionalidad de negocio con las capas de presentación, aplicación e infraestructura.

Aun con este proceso de modelado por DDD la solución aún se presenta con una arquitectura monolítica, debido a que toda la lógica de negocio aun forma parte de este compilado y como tal se publica en el mismo servidor junto con las demás capas, vistas y demás componentes. Lo que se ha logrado hasta este punto es mejorar la calidad del código, escalabilidad, reusabilidad, abstracción de las capas, dependencia mínima en la infraestructura, y mejora en el rendimiento del sistema, esto como resultado a que se ha logrado reducir la complejidad ciclomática, la duplicidad de código y se ha pasado de un entorno síncrono a asíncrono en la mayor parte de la comunicación entre capas.

Esta mejora en el rendimiento no se refleja en la reducción de los tiempos de respuesta en horas pico con alta transaccionalidad, como lo mencionamos anteriormente el sistema aun es monolito por lo que su lógica y el procesamiento de datos se hace bajo el mismo servidor, de igual formar otras aplicaciones alojadas hacen uso de los recursos.

Para mitigar esta situación proponemos una arquitectura distribuida de microservicios, la cual sugiere migrar los subdominios definidos en la capa de dominio a microservicios, debido a que estos cumplen con las propiedades de ser independientes, pequeños y manejan una sola funcionalidad.

**Clientes:** Los clientes son la aplicación web, que mantendrá la lógica de los CRUD y todo el módulo de administración del sistema y el servicio web que es expuesto a otras aplicaciones que necesitan emitir compensaciones. Para los clientes se planea dejar la

infraestructura tal como se encuentra en este momento, se extraerá la lógica y el acceso a datos de la aplicación, por lo que sólo serán encargadas de solicitar y renderizar la información al usuario. Esto reduce en gran medida el uso de los recursos de los servidores actuales. Estos clientes son el ICARE web y microservicios.

**Api gateway:** El api gateway será el encargado del descubrimiento y de manejar los endpoints de los servicios para que puedan ser consumidos, el proceso de registro de los servicios se hará automáticamente en el tiempo del bootstrap. Este api gateway será el encargado de orquestar los microservicios, será el único punto de entrada para los clientes y él será quien conozca los endpoints y redirija las peticiones entrantes al servicio correspondiente, así los clientes sólo tienen la necesidad de conocer un endpoint y no todos los endpoints de cada microservicio.

Se plantea una capa de seguridad Oauth2 para garantizar que sólo los clientes certificados puedan conectarse y consumir los servicios expuestos. Bajo esta propuesta todas las peticiones y servicios deberán ser bajo el protocolo REST y JSON como request y response, pero no se descarta la posibilidad de utilizar un protocolo de comunicación como SOAP.

**Microservicios:** Son el resultado de la identificación de los subdominios, contienen una única función de negocio y no dependen de otros servicios. Se planea un despliegue por medio de Docker y Swagger para la documentación y definir el api. Su consumo será bajo el protocolo REST y JSON como request y response. Cada microservicio contendrá una capa de acceso a datos ya sea a base de datos, texto plano, ftp, servicios web, entre otros.

Adicional a los subdominios identificados por la empresa y los subsecuentes microservicios obtenidos de cada uno de ellos, se agregan servicios para manejar las peticiones CRUD para enviar y recibir peticiones de la base de datos, así como un servicio para conversión de cantidades a moneda.

**Load balancer:** Se sugiere usar un balanceador para permitir la distribución de la carga en distintas instancias de forma transparente al momento de consumir el servicio. Muchas herramientas cloud permiten la creación automática de instancias de los servicios por medio de diferentes parámetros, por ejemplo, horas pico, uso de recursos, entre otros. Con esto se

garantiza que mientras las transacciones aumentan habrá un mejor manejo y escalamiento horizontal de instancias para atender la demanda.

**Productor/consumidor:** Se plantea usar la tolerancia a fallos para manejar de forma óptima los fallos o errores en los servicios, es decir, si un servicio falla esto evitará que se propague en cascada, de forma que podemos gestionar el error y controlar a nivel del servicio donde se produjo.

Por tal motivo se propone utilizar el patrón de diseño productor/consumidor para organizar las peticiones de los microservicios en colas que almacenen las peticiones ya sean SOAP, REST o JSON para que cada vez que un servicio lance una petición de actualización y/o validación de datos de datos, estas peticiones se almacenen en una cola esperando ser procesada por el servicio pertinente.

**Circuit Breaker:** Se plantea usar la tolerancia a fallos para manejar de forma óptima los fallos o errores en los servicios, es decir, si un servicio falla esto evitará que se propague en cascada, de forma que podemos gestionar el error y controlar a nivel del servicio donde se produjo. Por tal motivo se utilizarán llamadas a los servicios de validación y/o recuperación de datos para que se tome en cuenta un tiempo de espera para recibir la respuesta del microservicio encargado de devolver los datos y si este time out es alcanzado la petición se desestima por lo que el sistema se detendrá hasta recibir una nueva petición, esto se repetirá hasta que el servicio de validación y/o recuperación de datos este activo de nuevo.

Este patrón complementa a lo realizado en el patrón productor/consumidor, en dicho patrón las peticiones son almacenadas en una cola de peticiones, circuit breaker verifica en dichas colas de petición si la respuesta en forma de mensaje en la cola está presente, si no está presente, es decir si el time out de la petición es alcanzado, se hace una nueva verificación con un nuevo periodo de tiempo para ver si la respuesta está presente en la cola, de esta manera el sistema puede esperar de manera indefinida la respuesta en la cola y puede configurarse un límite de intentos para lanzar una advertencia de que los datos no están disponibles que puede ser configurada en el api gateway para 3 peticiones fallidas (que han alcanzado un time out), por ejemplo.

**Datos / aplicaciones legacy:** Se plantea reutilizar la base de datos transaccional del sistema, debido a que esta cuenta con alta disponibilidad y por lo tanto cada microservicio sólo envía una petición con las tramas de SQL a ejecutar, estas se almacenarán en una cola de peticiones en espera de que sean procesadas por la misma base de datos o por un manejador externo que redireccione estas tramas a la base de datos. No se propone crear una base de datos por cada microservicio debido a que involucra altos costos económicos y un proceso de migración de los datos a cada instancia. Los microservicios harán el consumo de los web services que manejan la información de los vuelos.

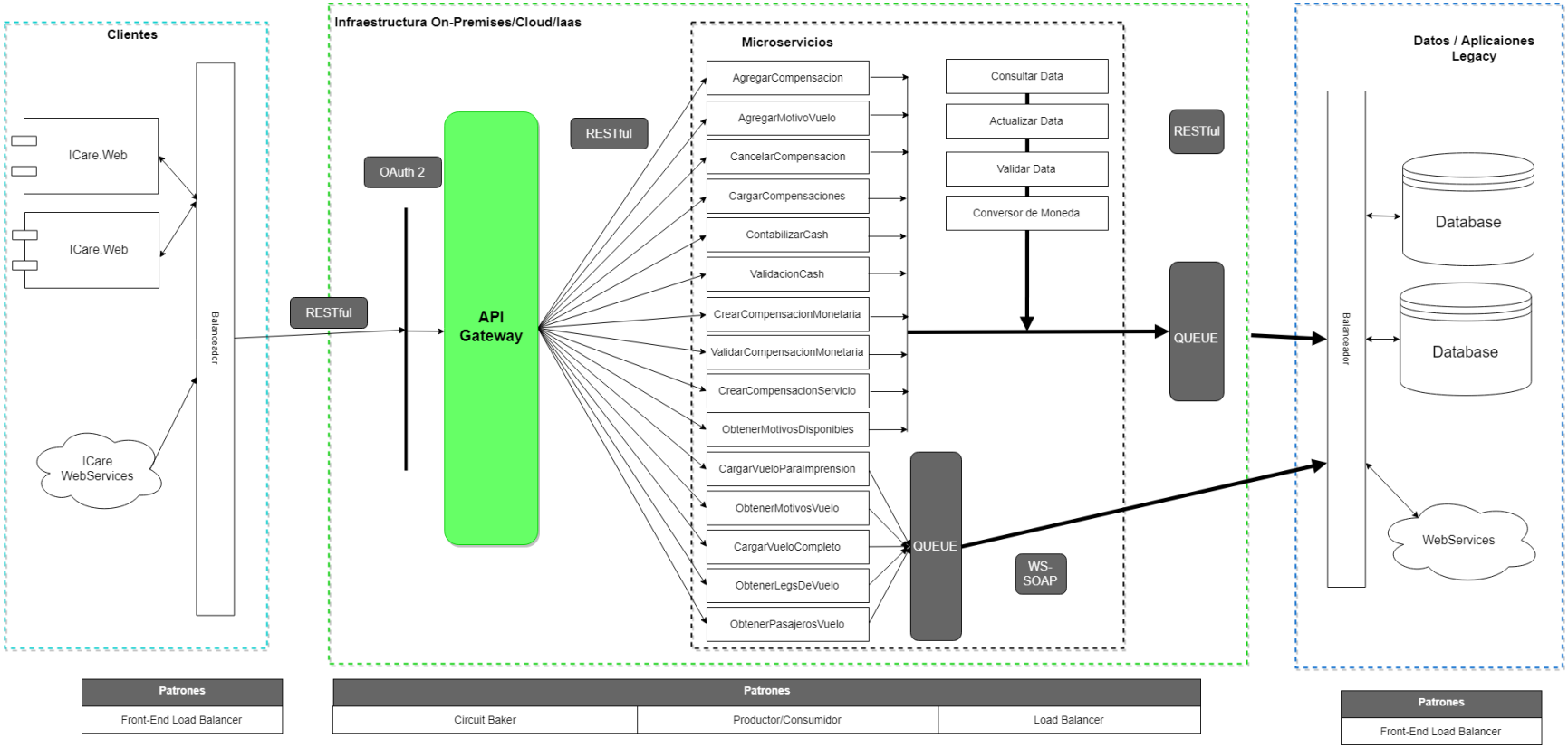


Imagen 15: Arquitectura de microservicios propuesta. Elaboración propia

## 5.2 Diagrama de capas para servicios

En el siguiente ejemplo se presenta una de las posibles composiciones de cada uno de los microservicios, nótese que no es necesariamente el diseño del producto final. En la propuesta arquitectónica presentada hemos sugerido que pueden utilizarse diferentes protocolos de comunicación, por ende diferentes tecnologías, en este ejemplo describimos el microservicio de agregar compensación cuyo modelo de dominio implementara la lógica para enviar y recibir las peticiones de las compensaciones de los vuelos, está esta implementada dentro de una tecnología JAVA y dichas peticiones son encapsuladas en peticiones JSON, para enviarse a una cola de peticiones, dentro de la lógica de cada servicio ´puede implementarse la lógica de chequeo de peticiones del patrón de diseño circuit breaker para chequear periódicamente las colas de peticiones/mensaje y/o la interacción con el api gateway dependiendo del servicio.

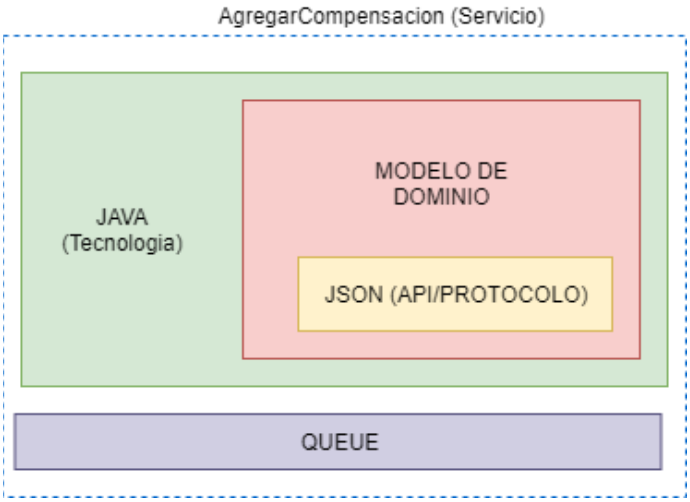


Imagen 16: Arquitectura de capa para un microservicio. Elaboración propia

Como se expresa anteriormente el ejemplo contempla utilizar Java como tecnología base donde será implementada la lógica del modelo de subdominio encapsulada dentro del microservicio, aunque no cierra las posibilidades a utilizar .Net, PHP, Angular JS u otra tecnología, de igual manera en el uso del protocolo y/o api pudiendo ser SOAP, HTTP, incluso SQL, la única restricción es que la cola de mensajería para manejar las peticiones soporte la trama de la petición utilizada

## 5.3 Diagrama de infraestructura

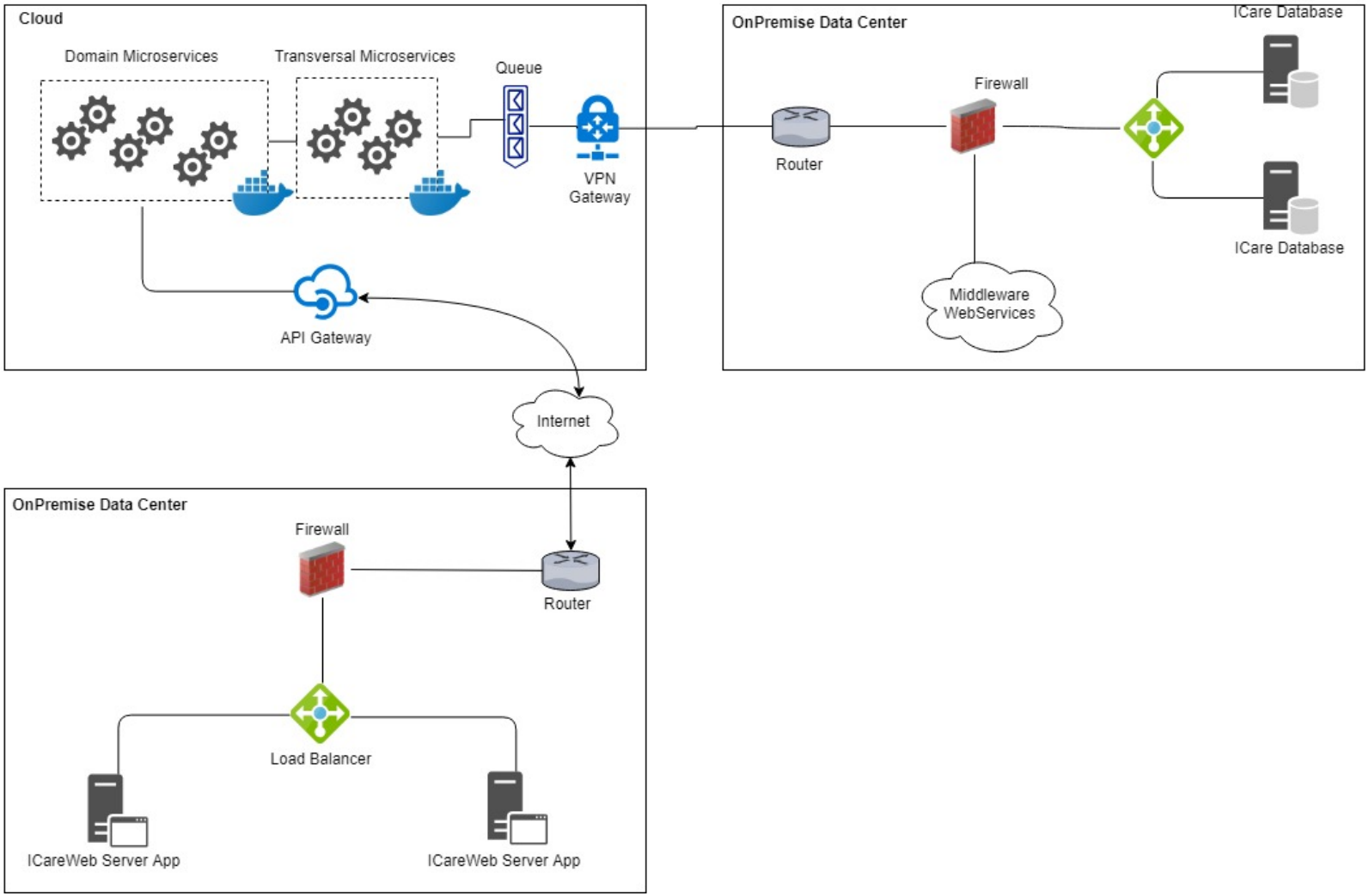


Imagen 17: Arquitectura de infraestructura. Elaboración propia

En esta propuesta se ha considerado implementar los servicios en la nube, actualmente existen diversos proveedores que ofrecen herramientas y servicios que ayudan a la administración, escalabilidad seguridad y monitoreo para este tipo de arquitecturas. Es a consideración de la compañía poder elegir el proveedor de servicios en la nube, debido a que puede estar sujeto a contratos marcos vigentes, cotizaciones de los servicios, licitaciones, entre otros puntos que la compañía estime.

Se recomienda hacer el despliegue de los microservicios en Docker, el cual nos da la flexibilidad de hacer el despliegue sin importar el lenguaje de programación o sistemas operativos ya sea Linux o Windows, y en servicios en la nube se puede hacer un escalado dependiendo de la demanda que se obtenga.

El api gateway es el canal de entrada que nos ofrece seguridad, enrutamiento y monitoreo de los microservicios. Este es accedido a través de internet por lo que el front-end podrá hacer llamadas directorios por ejemplo con AJAX.

En la infraestructura on-premise, se mantendrá el front-end del sistema y este deberá únicamente enviar peticiones al api gateway para obtener la información. Esto reducirá el costo de desarrollo, puesto que si se pretende hacer una migración completa hacia la nube es necesario hacer un replatform tanto en back-end como en front-end. Cabe mencionar que por el alto grado de demanda el front-end deberá permanecer en balanceador tal como se encuentra en la infraestructura actual.

Es importante considerar para una segunda fase la migración de la base de datos transaccional. En esta propuesta se ha considerado mantenerla en infraestructura on-premise, lo cual genera más latencia y a su vez un mayor grado de riesgo por si el servidor presenta fallas.

En esta propuesta los microservicios en la nube serán los encargados de consultar y operar contra la base de datos transaccional y para esto se necesita habilitar un túnel VPN para poder abrir la conexión entre ambas infraestructuras. El consumo del servicio para obtener información de los vuelos se hará mediante el mismo túnel VPN y será necesario crear reglas de firewall para que tanto la base de datos y el servicio de vuelos puedan ser consumidos por los microservicios.

# Capítulo 6: Guía de buenas prácticas para realizar un análisis domain driven design aplicado a microservicios

## 6.1 Introducción

La siguiente guía de buenas prácticas es un documento que pretende recopilar una serie de buenas prácticas que se pueden implementar en el desarrollo de un proyecto de migración y/o desarrollo de una aplicación basada en la arquitectura de microservicios. Esta guía se ha elaborado a partir de los resultados obtenidos del proceso de descomposición de dominio de una arquitectura monolítica usando DDD y del presente trabajo de graduación, el cual es una propuesta arquitectónica basada en microservicios para el sistema que es el propósito del caso de estudio y las opiniones de expertos con experiencia en proyectos de construcción o migración de aplicaciones usando una arquitectura basada total o parcialmente en microservicios.

Adicionalmente, esta guía ha sido construida únicamente en base a las entrevistas y recomendaciones que nos brindaron los expertos (Para más detalle ver transcripciones realizadas de las entrevistas que se tuvieron con los expertos, en el apéndice 1), y en base a la experiencia del caso de estudio en la empresa que se ha descrito anteriormente.

## 6.2 ¿Por qué utilizar microservicios?

Cuando se habla de crear una aplicación basada en una arquitectura de microservicios, lo primero que debemos contestar es ¿Tenemos claro que son los microservicios?

Principalmente una arquitectura de microservicios busca desacoplar los componentes de una aplicación para que estos sean individuales y sin dependencias, de manera que interactúen con otros componentes por medio de interfaces para lograr una funcionalidad dentro de nuestra aplicación. De esta manera podemos construir cada funcionalidad de nuestra aplicación como un servicio separado que puede interactuar con otros servicios para lograr una funcionalidad común y cada una de estas funcionalidades a su vez podemos descomponerla en servicios más pequeños que pueden ser reutilizables dentro de otras funcionalidades.

Una arquitectura de microservicios es una arquitectura distribuida y contempla cada funcionalidad de la aplicación como un conglomerado de uno o más servicios, no debe de confundirse con la arquitectura orientada a servicios, aunque los microservicios siguen la misma arquitectura, pero le añaden una dimensión extra en cuanto a la versatilidad y adaptabilidad que puede llegar a tener.

En un proyecto de migración o desarrollo de una aplicación de microservicios se debe tener en cuenta del porque se desea hacer este tipo de rediseño, todo proyecto parte de una idea la cual da lugar a una planeación y factibilidad de la misma para luego pasar a ser desarrollada, pero para hacerse una idea más clara de lo que implica el desarrollar una aplicación de microservicios, es más conveniente citar las desventajas de los microservicios para tener en cuenta los inconvenientes futuros que se pueden presentar durante el desarrollo o implementación del producto.

Tenga en cuenta que el utilizar una arquitectura de microservicios, conlleva una serie de ventajas en el desarrollo del proyecto, así como en la parte operativa del producto final.

- Los microservicios pueden distribuirse en varios contenedores de aplicaciones que pueden estar en varias redes informáticas.
- Los microservicios no dependen de una tecnología en particular, es más, en ocasiones puede darse el caso de que se mezclen tecnologías al mezclar varios microservicios que las implementen.
- El que los microservicios puedan agruparse en diferentes áreas de negocio y sean administrados por diferentes equipos puede liberar de carga de trabajo debido a que sólo tienen que atender a un área de negocio en particular y no necesitan desarrollar sobre otros microservicios de otras áreas. Newman (2015) dice que los microservicios pueden agruparse en módulos que pueden representar áreas diferentes del negocio y pueden interactuar entre, cada módulo estaría formado por servicios con un alto nivel de cohesión debido a que cada servicio obedece a la misma área de negocio.
- Las pruebas unitarias de los servicios pueden ser increíblemente rápidas y fáciles de diseñar y ejecutar.

- La tolerancia a fallos de las aplicaciones que implementen estos microservicios puede aumentar, dado que el fallo de un microservicio no condiciona que toda la aplicación deje de responder, habrán otros servicios que puedan sobrellevar hasta cierto punto el funcionamiento de la aplicación; sólo implicará que parte de la funcionalidad no esté disponible en ese momento y esto puede manejarse a discreción del diseñador de la aplicación (mensajes de error, cadenas de mensajes asíncronos, entre otros)
- Los microservicios pueden ser componentes reutilizables y aislados que puedan ser mantenidos independientes de cualquier otro sistema.
- La adaptabilidad de la aplicación permite incorpora y/o modificar nuevas funcionalidades de manera más dinámica sin arriesgar la integridad del sistema en sí.

Cuando no se empieza un proyecto de desarrollo o migración de un sistema informático, es lógico pensar que nosotros o quienes ponen en marcha el proyecto tienen una idea de cómo sobrellevar cada una de las fases del proyecto, así como las consecuencias positivas y/o negativas de llevar a cabo dicho proyecto, así como de su producto final. Pues bien, cuando hablamos de proyectos de desarrollo de aplicaciones informáticas con una arquitectura de microservicios, lo más probable es que en nuestro entorno nos encontremos con la dificultad de que puede haber una escases de recurso humano calificado con experiencia empírica y/o práctica en el desarrollo de este tipo de aplicaciones, más aún es posible que se nos dificulte encontrar desarrollos similares para tomarlos de referencia y como si esto no fuera suficiente, podemos darnos cuenta de que no contamos con la infraestructura y recursos necesarios para administrar y mantener una aplicación de este tipo.

Por tal motivo, el presente capítulo pretende mostrar una serie de recomendaciones que sirvan como referencia para abordar este tipo de proyectos, así como una serie de buenas prácticas que deberían considerarse no sólo en la ejecución de este tipo de migraciones, sino también en la planificación.

## 6.3 ¿Qué factibilidad existe en un proyecto de migración de una arquitectura monolítica a una de microservicios?

Cuando se piensa en las dificultades de manejar y mantener aplicaciones monolíticas, las cuales comúnmente son aplicaciones legacy en las empresas, el sólo pensar en poder transformar estas aplicaciones a arquitecturas y/o tecnologías vanguardistas como lo es microservicios que nos permitan no sólo mejorar la mantenibilidad y adaptación de nuestras aplicaciones, sino que también nos abren una gran gama de posibilidades para poder agilizar la competitividad de nuestras empresas, suena como una propuesta muy atractiva, pero no todo lo que brilla es oro, antes de embarcarnos en proyectos de migración de aplicaciones no sólo a una arquitectura de microservicios sino a cualquier tipo de arquitectura en general, debemos contestar una simple pregunta: "¿Se tiene la capacidad de llevar un proyecto de este tipo?", hablamos por supuesto de que deberíamos de realizar un estudio de factibilidad, pero un estudio de factibilidad colegialmente podemos decir que se compone de tres estudios diferentes, no es el caso de esta guía profundizar en qué es y cómo hacer un estudio de factibilidad, por lo que nos limitaremos únicamente a describir ciertos tópicos importantes que nos servirán de insumo para darnos cuenta si se tiene o no en la capacidad de ejecutar un proyecto de migración de arquitectura de una aplicación.

**Estudio de factibilidad económica**: Dicho de manera sencilla, es ver si se tienen o pueden conseguir los recursos económicos necesarios para llevar a cabo cada una de las fases del ciclo de vida de migración del proyecto, en este caso, es recomendable que este estudio debe de basarse al menos en una planificación preliminar, por lo que antes de empezar con un estudio de factibilidad debe de tenerse al menos una idea clara de qué es, cómo se llevará a cabo y cómo funcionará el entregable final de un proyecto, aunque es casi imposible conocer toda esta información a detalle en el inicio. Si es recomendable en la medida de lo posible buscar expertos asesores en el tema o al menos estudios y/o proyectos similares como marco de referencia.

**Estudio de factibilidad operativa:** Podemos decir que la factibilidad operativa de un proyecto comprende dos factores, primero que el proyecto se lleve a cabo de la manera planificada y el resultado final del mismo opere según los parámetros esperados, y segundo, que la empresa tenga los recursos necesarios tanto para llevar a cabo este proyecto para ponerlo en marcha.

**Estudio de factibilidad técnica**: Ésta comprende el estudio de varios tipos de factores técnicos que pueden afectar la ejecución de un proyecto. Además, estudia las posibilidades técnicas y tecnológicas de la capacidad instalada con la que la empresa cuenta o puede acceder para llevar a cabo el proyecto. También estudia las posibilidades de que existan factores externos que puedan afectar el desarrollo del proyecto o la implementación del entregable del mismo, ya sean de tipo legal o geográfico.

Si bien es cierto que los tres factores son importantes para determinar la factibilidad de un proyecto, para desarrollar la pregunta inicial de: "¿Que tan factible es poner en marcha un proceso de migración de una arquitectura monolítica a una de microservicios?" vamos a centrar nuestra atención en las factibilidades operativas y técnicas. Desafortunadamente en la mayoría de los casos se podría llegar a la conclusión de que no es muy factible ejecutar un proyecto de migración de una arquitectura monolítica a una de microservicios. Ahora se ve cómo podemos determinar esto, y para ello se propone que se deben de contestar al menos las siguientes seis preguntas:

- ¿Se cuenta con la capacidad instalada necesaria para llevar a cabo un proyecto de migración de esta envergadura?
- ¿Se tiene un equipo con experiencia necesaria para llevar a cabo un proyecto de migración de esta envergadura?
- ¿Se dispone del tiempo necesario y/o de planes de contingencia para llevar a cabo un proyecto de esta envergadura?
- ¿Se cuenta con el equipo humano necesario que le de mantenimiento correctivo y/o preventivo a la aplicación o grupo de aplicaciones resultantes?
- ¿La empresa está dispuesta a modificar algunos procedimientos internos dentro de la empresa para acoplarse al uso de la nueva arquitectura?
- ¿Por qué se desea migrar una aplicación monolítica a un esquema de microservicios?

Éstas y otras preguntas más deben de ser respondidas en el estudio de factibilidad operativa y técnica que se realice, y si alguna de ellas por alguna circunstancia no puede ser

respondida satisfactoriamente, entonces quizá deba pensar dos veces antes de iniciar un proceso de migración a una arquitectura de microservicios.

Es recomendable realizar el estudio de factibilidad económica al último momento, puesto que para realizar este estudio es necesario que se cuente con una planeación preliminar de cómo se ejecutará el proyecto y haber realizado los estudios de factibilidad operativa y técnica los cuales nos dará una mejor visibilidad de los pasos que se deben de considerar en el proceso de ejecución del proyecto.

Recuerde que parte de la factibilidad operativa es determinar en qué medida los procedimientos de la empresa pueden adaptarse a la implementación del proyecto de migración y el sistema resultante, mientras que la factibilidad técnica será la que determine si tenemos la capacidad instalada de llevar a cabo y mantener una aplicación de microservicios; estos factores deben de tomarse muy en cuenta a la hora de realizar un estudio de factibilidad económica por una razón muy importante, y es que en nuestra región por lo general los departamentos de informática o tecnología en una empresa son vistos como departamentos de servicio secundario, casi nunca se ven como departamentos estratégicos que puedan aportar un factor de diferenciación y competitividad al negocio; por estas razones lo más probable es que si la empresa donde se quiera ejecutar el proyecto no es muy grande y no cuenta con un buen presupuesto, se nos nieguen los fondos necesarios para el mismo.

Adicionalmente, con respecto al estudio de factibilidad operacional es que éste no debe simplemente limitarse a ver si los procesos de la organización se ajustan al desarrollo del nuevo proyecto, sino también en proponer de qué forma deben de adaptarse estos para que pueda llevarse a cabo tanto el proyecto de migración como la implementación del sistema resultante.

Tomemos como ejemplo la implementación de un ERP dentro de una empresa. Éste es de dominio popular en nuestro medio, la mayoría de las implementaciones de ERP para los procesos de la empresa usualmente resultan muy difíciles y costosos, pero ¿a qué se debe este comportamiento? Bueno, en gran medida se debe más a la resistencia de las organizaciones y personal a adaptar sus operaciones a los procesos propuestos por el ERP, usualmente se desea que el ERP sea el que se ajuste a las necesidades de la empresa y se gastan los recursos en tratar de hacer estas adaptaciones; si bien es cierto, esto es posible si las organizaciones fueran lo

suficientemente flexibles como para comprender que en muchas ocasiones es más recomendable adaptar los procesos de la empresa para interactuar mejor con las herramientas de la que dispone la organización.

Con respecto a la factibilidad económica del proyecto, tenga en cuenta que en las empresas quienes aprueban este tipo de proyectos no necesariamente son las personas que saben de tecnología, por tanto una de las consideraciones más importantes que se debe hacer es saber vender la idea, es decir, que dentro de la factibilidad económica es muy recomendable incluir los factores de recuperación de la inversión inicial, como el tiempo de recuperación de esta inversión y los beneficios que se generaran a partir del producto final del proyecto, por ejemplo, el aumento de productividad o de reacción a cambios y cómo esta solución beneficiará a nivel estratégico a la empresa. De tal forma se pueda presentar un análisis de costo/beneficio para el proyecto, un estudio de factibilidad económica puede mostrar que se cuentan con los recursos para llevar a cabo el proyecto.

## 6.4 Recomendaciones

### 6.4.1 Considerar el tamaño del servicio

Una consideración muy importante para tomar en cuenta es el tamaño que deben de tener los microservicios, esto puede llegar a condicionar la factibilidad técnica y operativa del proyecto y cambiar el enfoque a qué tipo de arquitectura es la más adecuada. Una buena práctica para seguir es la regla de “single responsibility principle”, esta regla indica que un servicio debe cumplir con una sola responsabilidad, pero se debe de tener en consideración que, si el número de servicios que se deben desarrollar es demasiado grande entonces la complejidad de nuestra aplicación también puede crecer a limites difíciles de mapear y mantener.

Se debe tomar en cuenta que un mayor número de servicios casi siempre demandará más recurso humano para mantenerlos y más recursos de infraestructura para albergarlos, si se colocaran todos los microservicios dentro de un miso contenedor, hasta cierto punto se está perdiendo una de las ventajas más grandes de este tipo de arquitectura, que es que pueden estar distribuidas en varios contenedores, reduciendo el riesgo de que exista una falla crítica del sistema al haber fallos en una de estas infraestructuras.

Otra forma de medición del tamaño puede ser el tiempo que nos tomaría rehacer este servicio en otra tecnología, y si es muy largo entonces deberíamos considerar partirlo. Tome en consideración que los microservicios pueden estar implementados con tecnologías diferentes, siempre y cuando tengan una interfaz común que les permita interactuar entre ellos, por ejemplo con una API (por sus siglas en Inglés, Application Programming Interface); por tanto queda a libertad de cada equipo de desarrollo elegir la tecnología que se utilizará para construir los microservicios, siempre y cuando ésta pueda implementar protocolos comunes de comunicación (SOAP, HTTP, REST, EDI, entre otros) que le garantice que pueda interactuar con otras entidades de su medio ambiente.

### 6.4.2 Considerar la cantidad de servicios

No existe una regla fija para definir qué tantos servicios serán necesarios para conformar nuestra aplicación, ya esto varía de aplicación a aplicación y de equipo a equipo, pero para determinar el número de servicios necesarios una consideración útil es tratar de buscar un equilibrio entre la complejidad y la manejabilidad, uno de los objetivos de una arquitectura basada en microservicios es proporcionar una gran agilidad y/o adaptabilidad a las aplicaciones, y esto se logra manteniendo los servicios de ésta lo más simples e independientes posible; no obstante, el mantener un servicio suficientemente simple como para que sea fácilmente adaptable y mantenible, depende en gran medida de la complejidad de los procesos que se implementen en su empresa y del sub proceso del que se encargue el servicio en cuestión.

Así mismo, se debe de tener en cuenta que agregar demasiados servicios sería un grave error debido a que agregaría una capa extra de complejidad para comprender los procesos de la aplicación, además que supondría una gran inversión en recursos de su empresa, es decir, de espacio en servidores, ancho de banda en la red, un equipo de trabajo que se encargue de él.; por otro lado, un número reducido de servicios podría suponer que la complejidad de los mismos es demasiado grande como para que se pueda considerar un microservicio y se busque seguir descomponiendo el subdominio donde opera ese servicio en servicios más pequeños, pero como lo vimos antes, se debe de tener en cuenta los recursos con los que se cuenta.

En conclusión, podemos decir que no existe un consenso de cuántos servicios deben de considerarse que tenga su aplicación para ser considerada bajo una arquitectura de microservicio; así, piense si el número de servicios que va a incluir proporcionará la agilidad que necesita mientras sigue siendo manejable para su equipo de administración.

### 6.4.3 Organizar los microservicios

Con el objetivo de mantener un orden dentro de la construcción de la arquitectura, una consideración muy útil es asegurarse de que cada microservicio esté asociado con un equipo de desarrollo y gestión dentro de su organización. Esto no significa que tenga que haber un equipo por cada microservicio.

Un sólo equipo puede manejar varios microservicios, y esto se hace para garantizar que la complejidad que tenga que manejar cada equipo sea la necesaria, es decir, no sobrecargar a los equipos de desarrollo, por ejemplo pueden agruparse los microservicios del área de finanzas, recursos humanos, operaciones, y cada uno puede ser administrado y mantenido por un equipo de desarrollo que tenga el conocimiento suficiente dentro del área en la que se desarrolla el microservicio a su cargo, de esta forma no sólo nos aseguramos de que los miembros de cada equipo de desarrollo sean expertos en su área, sino también nos aseguramos de que cada equipo sea conformado por el número de expertos suficientes para mantener los microservicios, entre estos desarrolladores, administradores de proyectos, analistas, personal de control de calidad, administradores de base de datos, administradores de servidores y redes. Recuerde que un equipo de trabajo relativamente pequeño es más fácil de organizar y puede ser más eficiente que un equipo de decenas o cientos de personas donde es más fácil que se presenten problemas que afecten la eficiencia equipo.

Otro objetivo que se busca con esto es simplemente asegurarse de que no se incluyan microservicios en su arquitectura que se conviertan en "huérfanos", porque nadie está disponible para desarrollarlos o soportarlos; o peor aún, microservicios que terminan siendo gestionados por múltiples equipos al mismo tiempo, lo que conduce a problemas de comunicación y organización.

### 6.4.4 Consumo de recursos

Cuando hablamos del consumo de recursos de los microservicios nos referimos a dos cosas distintas, primero debemos de considerar el recurso humano necesario para mantener el servicio y segundo a la cantidad de recursos tecnológicos que consumirán nuestro servicio, ya sea para alojarse o para interactuar, es decir, cuánto espacio consumirá en un determinado contenedor, cuanto ancho de banda se deberá reservar para este, o cuantas conexiones a la base de datos necesitará.

Muchos de estos factores serán el resultado de decisiones de diseño en los microservicios, pero es recomendable que se mantenga el número de recursos tecnológicos que consumirá el servicio como mínimo, entre más recursos necesite consumir el servicio más complejo se espera que sea este; por ende, un servicio menos mantenible o escalable, razón por la cual debería ponerse en tela de duda el estar utilizando una arquitectura de microservicios, recordemos una de sus características es de prestar una funcionalidad concreta y sencilla dentro de la aplicación, la cual sea independiente, aislada y escalable.

Adicionalmente los recursos tecnológicos necesarios para mantener los microservicios, también debe de considerarse la cantidad de recursos necesarios para su desarrollo. Para llevar a cabo un proyecto de desarrollo de microservicios es casi obligatorio una infraestructura de integración continua, y si se están utilizando tecnologías diferentes para desarrollar cada microservicio, muy posiblemente sea necearía más de una infraestructura, debido a que tendremos a más de un equipo trabajando en éstos y se harán constantemente pruebas de regresión e integridad; por lo que es recomendable tener herramientas que nos permitan automatizar aunque sea en parte los procesos repetitivos como verificaciones de código, administración de versiones, así como gestión de despliegues.

### 6.4.5 Revise el código y los procesos a refactorizar

Si estamos migrando una aplicación monolítica a una arquitectura distribuida de microservicios, es recomendable que los servicios se agrupen por unidades funcionales de la aplicación, esto es relativamente sencillo si su aplicación está construida en módulos funcionales

(independientemente que se trate de una aplicación monolítica o una distribuida), esto con el propósito de no tener que rediseñar o codificar más que lo estrictamente necesario, y así reducir el trabajo básicamente a las interfaces de comunicación de los servicios, la lógica funcional del código deberá permanecer intacta.

Si estamos refactorizando la lógica como microservicios utilizando la misma tecnología (por ejemplo, el lenguaje de programación Java), básicamente sólo tendremos que copiar y pegar el código en los nuevos servicios, y si estamos rehaciendo estos servicios en una nueva tecnología, el separar el código de las funcionalidades a migrar de manera separada nos ayudará a hacer una migración uno a uno entre el código de la aplicación original y los nuevos servicios.

### 6.4.6 La visibilidad de los servicios

Es recomendable mantener al mínimo la visibilidad de los microservicios que no deban interactuar de manera directa con el front-end de nuestra aplicación, de esta manera reducimos el riesgo de ataques a nuestra aplicación, también es recomendable mantener los servicios que contienen las APIs para el front-end y los servicios de back-end en contenedores separados si es posible siguiendo la filosofía de separar los servicios según la jerarquía organizacional que se tenga, de esta manera el que un servicio sea vulnerado no necesariamente condiciona que los demás servicios dentro de la aplicación lo sean, por ejemplo si tenemos un microservicio cuya función es acceder a los datos de usuario de la base de datos, sería recomendable que este servicio se accediera mediante otro(s) servicios que se invocaran desde el front-end en lugar de un servicio público que puede estar expuesto en un contenedor que no tenga mayores barreras de seguridad (como autenticación de usuario). Esto coincide con que los microservicios son por definición servicios aislados e independientes que solamente interactúan con los microservicios o interfaces estrictamente necesarias y no tienen visibilidad hacia otros servicios.

## 6.5 Herramientas y/o tecnologías necesarias

Antes de embarcarnos en un proceso de migración o creación de una aplicación que se construya sobre la base de una arquitectura de microservicios, es necesario primero verificar si se cuenta con la capacidad instalada para hacerle frente a un proyecto de esta envergadura.

A continuación, se presenta algunas tecnologías y técnicas útiles que le facilitarán el desarrollo de un proyecto de este tipo.

### 6.5.1 Integración continua (IC)

El utilizar una arquitectura de microservicios puede hacer que un proyecto de desarrollo se vuelva muy complejo si no se tiene la suficiente experiencia en este tipo de arquitecturas, pero esta complejidad puede ser reducida al utilizar un entorno de integración continua. Es muy recomendable que antes de iniciar un proyecto de microservicios se cuente con una infraestructura de IC la cual nos posibilita la configuración de pruebas automáticas y facilita los despliegues de los microservicios que se vayan desarrollando o ajustando a lo largo del proyecto, teniendo en cuenta que la cantidad de microservicios puede llegar a ser demasiado grande y este entorno puede ayudar a reducir tiempos y costos.

### 6.5.2 Patrones de diseño

Un patrón de diseño puede describirse como una solución para una situación lo suficientemente genérica como para qué ésta pueda aplicarse muchas veces sobre situaciones similares sin la necesidad de que se aplique de la misma manera. También podemos decir que un patrón de diseño es una solución a un problema de diseño principalmente de aplicaciones de software (aunque pueden utilizarse en otros ámbitos) que pretende resolver una determinada situación hasta convertirla en una configuración funcional cuya efectividad sea comprobable.

No es el objetivo de este documento el profundizar en los detalles de cada uno de los patrones de diseño existentes, sólo pretendemos mencionar la tipología de estos y señalar las características principales de los mismos para que el lector pueda seleccionar el o los tipos de patrones de diseño que podría utilizar para desarrollar los microservicios de su aplicación, dicho esto, podemos agrupar estos patrones de diseño en las siguientes categorías:

**Patrones creacionales**: Éstos hacen una abstracción de los objetos que se instancian dentro de una aplicación para hacer la creación, composición y representación de los objetos dentro de una estructura jerárquica de los mismos, que sea independiente de la funcionalidad de la aplicación. En conclusión, la utilidad de este tipo de patrones sobresale en las aplicaciones donde se utilice una tecnología orientada a objetos, por ejemplo, si construimos los microservicios con una tecnología como Java o C++.

**Patrones estructurales:** Los patrones estructurales se enfocan en cómo las clases y objetos se componen para formar estructuras mayores. Éstos describen como las estructuras compuestas por clases crecen para crear nuevas funcionalidades de manera de agregar a la estructura flexibilidad y que la misma pueda cambiar en tiempo de ejecución, lo cual es imposible con una composición de objetos estática; si bien es cierto que estos patrones de diseño se enfocan más en cómo las estructuras más simples se agrupan estructuralmente para formar estructuras más complejas que brinden funcionalidades mucho más complejas que a su vez nos pueden resultar útiles a pesar que su definición misma va en contra de la regla "single responsibility principle", la cual nos plantea que un microservicio debe de ser lo más pequeño posible en la medida que cumpla con una única funcionalidad. Estos patrones de diseño si bien no es recomendable utilizarlos para el diseño de los microservicios, si se pudieran utilizar para diseñar la interacción de los microservicios para formar una estructura de servicios que brinde una funcionalidad completa de la aplicación o para el módulo que se esté desarrollando.

**Patrones de comportamiento:** Los patrones de comportamiento se refieren a algoritmos y/o asignaciones de responsabilidades entre los objetos, un ejemplo de la utilización de uno de estos patrones podría aplicarse al diseño de menú de una aplicación, determinando la forma jerárquica de como las funciones de menú de la aplicación deberían de interactuar con los diferentes módulos de esta para brindar una funcionalidad. De igual manera que los patrones de diseño estructurales, este tipo de patrones son más útiles para determinar de qué forma deberían de interactuar nuestros módulos de servicio conformados por microservicios con el front end de nuestra aplicación, aunque también si la aplicación es suficientemente simple o los servicios suficientemente grandes, para describir como deberían interactuar estos, pero nuevamente, si se considera hacer servicios demasiado grandes se estará perdiendo la esencia de para qué se utiliza una arquitectura de microservicios.

Si bien es cierto que no es necesario utilizar ningún patrón de diseño para construir los microservicios, si es recomendable utilizarlos tanto para agilizar el trabajo, como para tener una filosofía de diseño uniforme entre los diferentes microservicios. como ya se mencionó anteriormente, es recomendable utilizar patrones creacionales para crear nuestros microservicios si se llegara a utilizar programación orientada a objetos, lo cual es lo más recomendable en nuestro medio este tipo de patrones y las tecnologías orientada a objetos son muy difundidas, mientras que por otro lado los patrones estructurales y de comportamiento son menos útiles en el desarrollo de los microservicios, pero pueden ser particularmente útiles para que un arquitecto de software encargado de definir la interacción entre los servicios pueda seguir uno de estos patrones para diseñarlos.

### 6.5.3 SOAP

Como ya lo hemos mencionado, los microservicios pueden tomarse como una evolución de la arquitectura orientada a servicios, aunque con una gran diferencia, el tamaño de los servicios es mucho menor, es decir que a diferencia de los servicios en una arquitectura SOA, en una arquitectura de microservicios cada uno de los servicios realiza una única acción dentro del subdominio de la aplicación. Por tratarse de servicios, una posibilidad es que utilicemos SOAP (Simple Object Access Protocol) para construir nuestros servicios, no es el objetivo de este documento profundizar en el uso de SOAP, pero lo recomendamos como una tecnología a utilizar porque en nuestro medio SOAP es la tecnología más comúnmente utilizada

### 6.5.4 BPM/BPMS

Business Process Management no es una tecnología en sí, es una disciplina de gestión de procesos de negocio y tecnologías de las empresas cuyo principal propósito es mejorar la eficiencia de los procesos de negocio y su capacidad para configurarse de la manera más conveniente según las necesidades del negocio, pero porque hacemos mención de ella entonces, la respuesta es relativamente sencilla aunque su implementación no lo es tanto, si se logra implementar satisfactoriamente una arquitectura de microservicios aunque sean una única

aplicación dentro de la organización, se estaría dando el primer paso para poder implementar una disciplina de administración de procesos de BPM o mejor aún una aplicación BPMS que implemente algún lenguaje de muy alto nivel como BPEL que permita administrar la configuración de los servicios en tiempo real, aunque claro esto dependerá en gran medida de que tan granular es la funcionalidad que ofrecen los servicios y que tan independientes y genéricos son, puesto que una aplicación BPMS nos permitiría organizar estos microservicios para ofrecer servicios y/o funcionalidades nuevas basadas en las funcionalidades existentes de los microservicios que permitan a los procesos de negocio de la aplicación y por ende de la empresa adaptarse más ágilmente a las exigencias del macro negocio de las cuales son una herramienta facilitadora.

Anteriormente dijimos que en nuestro entorno difícilmente se ve a las empresas como unidades estratégicas que otorguen valor agregado a la organización, los departamento de IT, tecnología, informática o como quiera que se llamen se perciben como simples departamentos de servicio de manejo de datos/información para otros departamentos de la organización que se consideran más importantes (ventas, adquisiciones, finanzas, entre otros.), pero la implementación de este tipo de herramientas de BPMS pueden agregar un gran valor estratégico al negocio agilizando la adaptabilidad de los procesos del mismo, el dar a conocer este tipo de iniciativas dentro de la empresas podría a ayudar a cambiar la percepción que se tiene de nuestros departamentos de informática, aunque aclarando que la implementación de estos requieren recursos y esfuerzos no serían capitalizables sino hasta el largo plazo.

### 6.5.5 Domain driven design

Podemos decir que el dominio refleja el comportamiento y el modelo de la aplicación, en el cual se definen los procesos y las reglas de negocio sobre las cuales ha sido construida la aplicación y que domain driver design es un enfoque para el desarrollo de software definido por Evans (2003) y es la técnica que pretendemos utilizar para crear los microservicios que integren nuestra aplicación.

Según domain driven design nosotros podemos tomar el dominio de una aplicación y aplicar una serie de metodologías para expresar este dominio en subdominios de una manera iterativa con la intención de obtener subdominios suficientemente pequeños que puedan ser transformados en los microservicios de nuestra aplicación.

## 6.6 Como realizar una descomposición con domain driven design

"Un modelo es una simplificación, es una interpretación de la realidad que abstrae los aspectos relevantes para resolver un problema ignorando detalles extraños", Evans (2003), página 6. Como ya lo mencionamos domain driven design es un concepto que podemos utilizar para definir los microservicios de nuestra aplicación, partimos de la premisa que un modelo es una abstracción de la realidad y como su nombre lo indica pretende dirigir el diseño y desarrollo de una aplicación, partiendo de la descomposición que se haga del dominio, es decir que el dominio de la aplicación se descompone en subdominios más pequeños que interactúan entre si hasta llegar a formar nuestra aplicación, es de esta descomposición en subdominios donde nosotros encontraremos los insumos necesarios para crear los microservicios. Para ello describiremos los pasos a seguir para poder realizar un modelado de descomposición de dominio y a partir de allí podremos empezar a diseñar los microservicios utilizando cualquier técnica de diseño que creamos conveniente.

### 6.6.1 Lenguaje ubicuo

La primera idea central de domain driven design es la creación de un lenguaje ubicuo que sea común a todos los involucrados en el desarrollo del sistema, esto incluye a personal técnico y usuarios de negocio que son los expertos de dominio.

El lenguaje ubicuo no debe de considerarse como un paso o característica exclusivo de domain driver design, si bien es cierto que en la gran mayoría de referencias bibliográficas que se pueden hallar acerca del tema se menciona como una de las 2 ideas fundamentales del análisis, el lenguaje ubicuo es sólo el primer paso en la integración del departamento de informática hacia un nivel donde se pueda guiar el crecimiento estratégico del negocio, apoyado

en las ventajas que puede suponer la implementación de la tecnología, es decir puede llevar a la implementación de filosofías y técnicas más como COBIT 5 o BPM, por tal motivo la creación del lenguaje ubicuo no es una tarea que se deba de tomar a la ligera si se tienen una visión de llevar al departamento de informática más allá de un simple departamento de servicios técnicos dentro de la empresa o negocio.

La mejor manera de empezar a construir el lenguaje ubicuo es reunir a todos los involucrados en el desarrollo del sistema para que cada miembro empiece a relatar sus experiencias y responsabilidades, así como su interacción con el sistema y otras acciones dentro del dominio de la aplicación, esto se hace con el fin de que el equipo técnico empiece a familiarizarse con la terminología utilizada en el negocio y con el fin de aclarar cualquier tipo de dudas que puedan existir. Uno de los primeros pasos en la fase de desarrollo de un proyecto es la recolección de requerimientos funcionales y no funcionales de la aplicación, en este caso nos centraremos en los requerimientos funcionales, una de las funciones de los analistas de sistemas o de quien sea que esté a cargo de esta fase en el desarrollo es servir de traductor entre los requerimientos expresados en lenguaje de negocio para entregárselos al equipo de desarrollo, pero si este traductor llega a cometer un error, lo más probable es que el error se detecte hasta cuando se implemente la solución o en el peor de los casos cuando esta esté en producción. Cuando hablamos de malentendidos entre los expertos de dominio nos referimos a que cada experto tendrá sus propias experiencias y es posible que se refieran a situaciones similares de la misma manera, dando lugar a confusión a la hora de recolectar requerimientos, por ellos es importante deshacerse de estos mal entendidos antes de empezar con la fase de desarrollo.

Por lo dicho anteriormente, se podría pensar que el crear un lenguaje ubicuo es simplemente hacer del dominio del equipo de desarrollo la terminología utilizada por los miembros del equipo del negocio y en el caso más sencillo esta es la mejor aproximación de lo que podría ser un lenguaje ubicuo, no obstante si es posible obtener una terminología común a ambos equipos se reduciría aún más el riesgo de que existan malos entendidos de comunicación al recolectar los requerimientos funcionales para el desarrollo y diseño del sistema.

### 6.6.2 Descomposición del dominio

En la industria de TI, muchas metodologías han planteado el hecho de hacer análisis y diseños separados de un sistema a partir de un dominio que pertenece a un dominio más grande, model driver architecture (MDA) y el platform specific model (PSM) son ejemplos de ello y esta es precisamente la idea central de domain driver design, la descomposición de un dominio más grande en subdominios que sean más manejables, entendibles y escalables. Como hemos dicho un dominio es una representación de un sistema una simplificación del mismo la cual nos puede servir para representar resolver un problema en este caso de diseño, un dominio objetos de dominio y reglas de negocio y la interacción entre estos objetos, este modelo de dominio es el que se construye apoyándonos en las definiciones establecida en el lenguaje ubicuo, de esta forma los expertos de dominio y los analistas de sistema serán los encargados de crear este modelo de dominio el cual será utilizado por los diseñadores o arquitectos e software para construir un modelo de implementación que servirá como guía para los desarrolladores de la aplicación, en este caso de cada uno de los microservicios que serán la traducción de cada uno de los subdominios que describen la aplicación, de allí el nombre “Diseño guiado por el dominio”. Este modelo de implementación usualmente se representa utilizando UML (universal model lenguaje) y se compone por las dependencias entre las partes de código del sistema (diagramas de componentes) y la estructura del sistema en ejecución (diagrama de despliegue).

Por último se señala que el proceso de descomposición de dominio que se hace sobre el sistema es un proceso iterativo y el número de iteraciones necesarias será determinado por los criterios de selección del tamaño de los microservicios, por tanto se debe de tener una idea clara del alcance de la solución que se desea diseñar con el objeto de que al momento de realizar la descomposición del dominio obtengamos un numero de subdominios que den lugar a la creación de un numero de microservicios que sean sostenibles por la infraestructura con la que cuenta la empresa y el equipo de trabajo que se tenga disponible para mantenerlos.

Recapitulando, para aplicar el concepto de domain driven design debemos de seguir los siguientes pasos:

- Conformar el equipo de trabajo necesario, esto debe de incluir a expertos de dominio y al equipo de IT encargado de llevar a cabo el proyecto, esto debe de incluir a desarrolladores, analistas de sistemas y arquitectos de software.
- Construir nuestro lenguaje ubicuo a partir de las experiencias narradas de los expertos de dominio, esto puede hacerse reuniendo a los expertos de dominio y solicitándoles compartan su apreciación y experiencia del dominio frente a los demás expertos y personal técnico, esto con el propósito de despejar dudad en caso de que los expertos de dominio tengan opiniones y experiencias discrepantes con respecto a lo que el dominio debe ser, también en la medida de lo posible los miembros técnicos pueden colaborar a redefinir conceptos que sean útiles para describir el dominio tanto para los expertos de dominio como para los usuarios técnicos.
- Documentar el lenguaje ubicuo que se construyó a partir de las reuniones con los expertos de dominio y el personal técnico, tenga en cuenta que este lenguaje puede cambiar en el tiempo en la medida de que se profundice en los subdominios, pero en la medida de lo posible deben de abordarse el mayor número de definiciones desde el inicio para evitar retrabajos y malentendidos en las fases diseño y desarrollo.
- Construir el modelo de dominio a partir de las definiciones establecida en el lenguaje ubicuo.
- Descomponer el dominio en subdominios más pequeños que puedan ser manejables, este proceso es iterativo y se descompondrán hasta encontrar a lo que a juicio o conveniencia del equipo de desarrollo y diseño sean las expresiones más pequeñas del dominio de la aplicación, de allí se construirán los microservicios. Tenga en cuenta de que al descomponer cada dominio en subdominios estos deben de ir agrupándose según las áreas de negocio las jerarquías que se tengan definidas.
- A partir de cada subdominio se debe de modelar el modelo de dominio y a partir de este el modelo de implementación, usualmente este se puede definir con UML como diagrama de componentes y diagrama de despliegue, aunque esto no es estrictamente obligatorio.
- Una vez todo este hecho puede pasarse a la fase de desarrollo, en esta fase cada subdominio que serán expresado como microservicios serán desarrollados por los

equipos de trabajo de cada área, cada uno utilizando un entorno de integración continua, puede ser el mismo si se utilizan tecnologías compatibles, pero esto no es obligatorio.

## 6.7 El equipo de trabajo

No podemos embarcarnos en un proyecto de construcción o migración de una aplicación a microservicios si no contamos con el personal idóneo que pueda llevar acabo esta encomienda, lógicamente los miembros del equipo de trabajo deberán ir acorde al recurso humano planificado.

### 6.7.1 Competencias del equipo de trabajo

Por lo que hemos hablado hasta el momento ya deberíamos tener una idea de las competencias que deben de tener los miembros de nuestro equipo de trabajo, pero tenga en cuenta que esta va a depender de las tecnologías y técnicas que decida emplear en su proyecto, acá planteamos el realizar esta migración a partir de un modelo descompuesto de dominio mediante domain driven design.

Obviamente lo primero que se desea buscar en el perfil de quienes puedan integrar el equipo de trabajo es: “tener experiencia en desarrollo de aplicaciones de microservicios”, pero honestamente en nuestra región son escasas las personas que tengan experiencia en el ramo, a lo sumo se podrá encontrar personas que tengan una noción general del tema, por lo que es más recomendable de buscar personas que estén dispuestas a experimentar y aprender, aparte de las consideraciones deseables que se desea de un colaborador, que sea proactivo, sepa trabajar en equipo, acostumbrado a trabajar bajo presión, creativo, ordenado, alguno de estos factores de conocimiento a tomar en cuenta según el rol de cada miembro dentro del equipo son:

Coordinador de proyecto:

- ☑ Planeación de proyectos, por ejemplo, PMI
- ☑ Gestión de riesgos (planes de contingencia)

Arquitecto de software:

- ☑ Patrones de diseño creacionales, estructurales y de comportamiento
- ☑ UML
- ☑ Diseño de redes

Desarrollador de software:

- ☑ Patrones de diseño creacionales
- ☑ SOAP/REST
- ☑ Protocolos de hipertexto basados en XML
- ☑ La tecnología seleccionada, Java, C++, Visual Basic, entre otros.
- ☑ Manejo básico de aplicaciones de IC

Administrador de aplicaciones:

- ☑ Manejo de aplicaciones de IC, por ejemplo, Jenkins, SonarQube, GIT, entre otros.

Tester de software:

- ☑ Creación y uso de casos de prueba
- ☑ Conocimiento de pruebas automatizadas
- ☑ Pruebas de regresión

# Capítulo 7: Conclusiones y recomendaciones

Durante el desarrollo de este estudio se pudo constatar bajo un escenario real, que una aplicación diseñada con una arquitectura monolítica puede enfrentar dificultades cuando el tamaño del código fuente crece y son constantes sus requerimientos o mejoras, por ende, el desarrollo comienza a ser cada vez más difícil y aumenta la complejidad de entendimiento, acoplamiento, mantenibilidad y escalabilidad.

A continuación se presentan las conclusiones en relación a los objetivos de la investigación:

**Identificar los componentes del dominio que se migrarán a microservicios distribuidos.**

En el análisis de este caso de estudio se logró comprender el concepto de domain driven design definido por Eric Evans, así como también los aspectos a tomar en cuenta para poder aplicarlo en un proyecto real, tal es el caso de la definición de un lenguaje ubicuo, la conformación del equipo de trabajo, como identificar el dominio de una aplicación, el proceso de descomposición de dominio y como deben ser definidos los subdominios.

La arquitectura de domain driven design es fundamental para poder separar la lógica de negocio de la funcionalidad transversal y la capa de presentación de un sistema, y de esta forma se ha logrado identificar y aislar los subdominios que contienen la lógica de negocio en una capa de dominio la cual no depende ninguna otra capa. Es decir que se logró reestructurar la arquitectura monolítica de una definición vista – lógica – datos a una arquitectura monolítica basada en domain driven design desacoplada en su lógica de negocio.

**Definir una arquitectura distribuida usando microservicios partiendo del modelo de dominio.**

Partiendo de la arquitectura monolítica resultante luego de aplicar la descomposición según domain driven design, se construyó una propuesta arquitectónica distribuida basada en microservicios. Para esto se logró comprender el concepto de una arquitectura de microservicios, así como la definición de un microservicio, sus características y como operan unos con otros en un ambiente distribuido.

La propuesta creada considera migrar los subdominios identificados con domain driven design a microservicios, ya que estos se definieron y construyeron siguiendo el patrón de responsabilidad única, es decir, son dueños de una única funcionalidad de negocio, además de agregar microservicios transversales para las operaciones de manejo de datos y otras funcionalidades particulares. Para llevar a cabo esta arquitectura se ha sugerido el aprovisionamiento de los recursos en la nube, ya que esta provee un alto grado de seguridad, escalabilidad, mantenibilidad, control y seguimiento de los recursos y es capaz de adaptarse a altas demandas creando nuevas instancias de cada microservicio en caso se requiera.

En el proceso de construcción de la propuesta se tomó en consideración el desacoplamiento de las funcionalidades técnicas, de negocio y de presentación resultante de la primera fase de esta investigación, y como resultado se sugiere mantener el front-end en infraestructura on-premise y toda la funcionalidad técnica y de negocio en infraestructura en la nube, así mismo se dan los lineamientos para la seguridad, conectividad y tolerancia a fallos de manera que se garantice la operatividad de la aplicación.

**Elaborar un documento de recomendaciones para la migración de un sistema monolítico a un sistema distribuido.**

Al enfrentar un proyecto de migración de arquitecta existen consideraciones a tener en cuenta que no todos logran identificar en un inicio, ya que puede significar un alto grado de complejidad y no en todas las aplicaciones puede considerarse una migración de arquitectura.

Como resultado de esta investigación se ha construido un documento con recomendaciones que puede servir a cualquier desarrollador, o equipo de TI que tenga un proyecto de esta naturaleza. Dichas recomendaciones fueron construidas en base a la investigación documental, entrevistas y experiencias propias que llevo la construcción de este trabajo. Por ello plasmamos los siguientes puntos a considerar.

A. El primer paso para poder iniciar este tipo de migración es enfocarse en el factor humano. Puesto que trabajar con microservicios primeramente es un cambio de conceptos y mentalidad en el área de IT, de una forma lineal y estructural a una forma distribuida, y en segundo lugar que entiendan muy bien los conceptos involucrados en esta nueva

arquitectura de sistemas. Para este tipo de migraciones y desarrollo de microservicios, se requiere de un equipo de desarrollo capacitado y que mejor se adapte a los cambios.

B. Establecer un marco común y acordado utilizando domain driven design, y con ello entender cuáles son los actores y procesos que están involucrados en el entorno del sistema, cuáles procesos se verían afectados en la migración e identificar los subdominios, y a partir de estos se podrán identificar los nuevos microservicios.

C. Se debe entender y definir el tamaño y el alcance del microservicio, debido a que un microservicio debe tener una responsabilidad única dentro del marco de las necesidades del negocio.

D. No empezar a diseñar todos los procesos de la empresa como microservicios, sino que empezar con pocos y que no sean de gran impacto para la empresa. A medida que se vaya comprendiendo dicha arquitectura y el proceso de migración, se irán incluyendo acorde a la necesidad de la empresa más procesos a la migración.

E. Tener en cuenta que, al montar una arquitectura de microservicios, se recomienda implementar a futuro un flujo de integración continua, con el objetivo de facilitar la administración y despliegues de estos microservicios.

F. Se recomienda trabajar desde un inicio con un marco de trabajo ágil para llevar a cabo la migración de un sistema monolito a un sistema distribuido con microservicios, debido a que las necesidades del negocio van cambiando día con día, y la solución debe apegarse a dichos cambios.

G. Se recomienda definir y tener claridad de los requerimientos, hacer un buen diseño arquitectónico de la solución, previo al desarrollo.

## Referencias bibliográficas


D .S. Behar Rivero (2008), Metodología de la Investigación, Editorial Shalom, ISBN 978-959-212-783-7

S.,l. Tuylor· y H. Bogdun (1994), Introducción a los métodos cualitativos de investigación, ediciones PAIDOS, Barcelona, ISBN: 84.·7509·816·9

R. C. Martin and M. Micah (2006), Agile Principles, Patterns and Practicles in C#, Prentice Hall, ISBN-13 978-0-13-185724-4

L. Bass P. Clements y R. Kazman (2012), Software Architecture in Practice 3th Edition, Addison-Wesley, Westford, Massachusetts, ISBN 978-0-321-81573-6

A. Lafuente (2012), Introducción a los Sistemas distribuidos, Departamento de Arquitectura y Tecnología de Computadores, UPV/EHU.

Coulouris G., Dollimore J., Kindberg T., Blair, G. Distributed Systems: Concepts and Design, 5ta edición. Upper Saddle River, Nueva Jersey: Pearson Education, Inc. ISBN-13: 978-0132143011.

C. B. Reynoso (2004). Introducción a la Arquitectura de Software. UNIVERSIDAD DE BUENOS AIRES

D. Haywood (2009). Domain-Driven Design, the Pragmatic Book self, Raleigh, North Carolina Dallas, Texas. ISBN-13 978-1-934356-44-9

E. Evans (2015) Domain Driven Design: Reference Definitions and Pattern Summaries. 1ra edición, Pearson Education, Inc. Boston, Massachusetts. ISBN- 978-1-4575-0119-7

E. Evans (2003). Domain-Driven Design: Tackling Complexity in the Heart of Soft, 1ra edición. Boston, Massachusetts: Pearson Education, Inc. ISBN-13: 978-0321125217.

E. Gamma, R Helm, R. Johnson, J. Vlissides. (1994) Design Patterns: Elements of reusable object-oriented software, Allyson Wesley, Hawthorne, New York. ISBN 978-0201633610

D. Garlan y M. Shaw (1994). An Introduction to Software Architecture... CMU-CS-94-166. School of Computer Science. Carnegie Mellon University. Pittsburgh, PA 15213-3890.

F. Bachmann, L Bass, P. Clements, D. Garlan, J. Ivers, R Little, Robert Nord, and Judy Stafford (2011). Documenting Software Architectures: Views and Beyond, 2da Edición, Pearson Education, Westford, Massachusetts. ISBN-13: 978-0-321-55268-6.

G. Lenz y C. Wienands (2006), Practical Software Factories in .NET. 1ra edición. Springer-Verlag New York Inc. ISBN-13: 978-1-59059-665-4.

M. Flowers, J Lewis. (2014). Microservices. 25 March 2014, de MartinFlowers.com Sitio web: http://martinfowler.com/articles/microservices.html

S. Millett (2015). Patterns, Principles, and practices of DDD, 1a edición, John Wiley & Sons, Inc. Indianapolis, Indiana. ISBN- 978-1-118-71470-6.

I. Nadareishvili, R. Mitra,M. McLarty ,M Amundsen (2016) Microservice Architecture: Aligning Principles, Practices, and Culture, 1ra edición. Sebastopol, Ciudad de California: O'Reilly Media, Inc. ISBN-13: 978-1491956250.

S. Newman, (2015). Building Microservices: Designing Fine-Grained Systems, 1ra edición. Sebastopol, Ciudad de California: O'Reilly Media, Inc. ISBN-13: 978-1491950357.

M. Richard (2016), Microservices Vs. Service Oriented Arquitecture, primera edición, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc ISBN 978-1-491-94161-4

N. Rozanski, E. Woods (2011). Software Systems Architecture: Working with Stakeholders and viewpoints, 2da Edicion, Allysn Wesley, Westford, Massachusetts. ISBN-13: 978-0-321-71833-4.

F. Oquendo, J. Leite y T. Batista (2016). Software Architecture in Action: Designing and Executing Architectural Models with SysADL Grounded on the OMG SysML Standard (Undergraduate Topics in Computer Science), 1ra edición. Springer. Switzerland. ISBN-13 978-3319443379

M. Rosen, B. Lublinsky y K. Smith (2008). Applied SOA: Service-Oriented Architecture and Design Strategies. 1ra edición. Wiley Publishing, Inc. Indianapolis. ISBN-978-0-470-22365-9.

I. Sommerville (2012). Ingeniería de Software, 9na edición. Naucalpan de Juárez, Estado de México: Pearson Educación de México, S.A. de C.V. ISBN-13: 978-6073206037.

I. Sommerville (2015). Software Engineering, 10ma edición. England: Pearson Education Limited. ISBN-13: 978-0133943030.

R. Stephens (2015). Beginning Software Engineering, 1ra edición. Indianápolis, Indiana: John Wiley & Sons, Inc. ISBN-13: 978-1118969144.

Stephen D. Burd (2011). Systems Architecture, Sixth Edition, Boston, MA. ISBN-13 978-0-538-47533-4

A. S. Tanenbaum., V. Steen (2006). Distributed Systems: Principles and Paradigms, 2da edición. Upper Saddle River, Nueva Jersey: Pearson Education, Inc. ISBN-13: 978-0132392273.

V. Vernon (2013), Implementing Domain Driven Design, 1a edición. Pearson Education, Inc. Westford, Massachusetts. ISBN-13: 978-0-321-83457-7

E. Wolff (2016). Microservices: Flexible Software Architecture, 1ra edición. Reading, Massachusetts: Addison-Wesley. ISBN-13: 978-0134602417.

G. G. Maigua (2012), Buenas Prácticas en la Gestión y Dirección de Proyectos Informáticos, Facultad Regional Tucumán Universidad Tecnológica Nacional – U.T.N. Argentina ISBN: 978-987-1896-01-1

PMI (2013), GUÍA DE LOS FUNDAMENTOS PARA LA DIRECCIÓN DE PROYECTOS 5ta Edición, Dirección de Proyectos. I. Project Management Institute. II. Título: Guía del PMBOK. ISBN 978-1-62825-009-1

F. G. Meza (2015), Introducción a la Ingeniería Industrial 1ª Edición, Universidad Continental Jr. Junín 355, Miraflores, Lima-18

M. S. Figueroa (2001), Gestión integrada de proyectos, Edición de la Universidad Politécnica de Catalunya, SL, ISBN: 84-8301-453-X

D. Krafzig, K. Banke, D. Slama (2004), Enterprise SOA: Service-Oriented Architecture Best Practices (Coad), Prentice Hall, First Edition, ISBN-13: 978-0131465756

## Apéndices

### Apéndice 1. Entrevistas a expertos

En este apartado se encuentra las entrevistas a los expertos consultados durante el desarrollo de esta investigación.

**Experto 1**

**Nombre de Experto: Max Frank Rodriguez.**

**Nacionalidad: colombiano.**

**Área de campo: Arquitecto empresarial y consultoría.**

*Entrevistador:*

Desde su punto de vista, porque se trata de migrar una arquitectura monolítica a una arquitectura distribuida, en este contexto, porque que no siempre es conveniente llevar una aplicación de una arquitectura monolítica a una distribuida. Es más, la arquitectura monolítica tiene ciertas ventajas dependiendo para que y donde se utilice la aplicación.

Desde la perspectiva de un arquitecto de software, porque razones busca llevar una aplicación monolítica a una distribuida

*Entrevistado:*

La principal ventaja por lo que se hace la migración es “Time To Market”. Cuando se agarra una aplicación monolítica y se quiere hacer un cambio, por ejemplo, colorear un botón en una aplicación monolítica frente a una distribuida costaría tiempo de desarrollo, en una aplicación distribuida donde independientemente el botón se utilice en una clase o varias el desarrollo costaría lo mismo, la ventaja estaría en: Procesos de control de cambios.

Cuando en una aplicación distribuida el botón esta como un microservicio, sólo habría que probar ese microservicio, siendo 100% optimistas, si toda la aplicación está distribuida utilizando microservicios, no habrá necesidad de probar todo el conjunto de componentes, posiblemente sólo se pruebe el componente individual bajo dos factores, que funcione y que falle. Recuerde que una característica que deben de cumplir las aplicaciones construidas con microservicios es que sea tolerante a fallos, es decir

que, si por alguna circunstancia un microservicio falla, en teoría la aplicación deberá ser capaz de seguir operando, no al 100: porque algo fallo.

Si tuviéramos una aplicación 100% utilizando microservicios, las pruebas de estos microservicios deben de ser más rápidas que si probáramos toda la aplicación. Entonces lo importante es la velocidad de prueba, como experiencia personal en una aplicación monolítica toma 4 o 5 semanas de prueba más 1 o 2 semanas de control de cambios, porque hay que probar toda la aplicación para comprobar que no exista nada que pueda fallar, además en el control de cambios se tiene que pedirle permiso a 4 o 5 áreas diferentes, en una aplicación distribuida todo este proceso se podría hacer más rápido. Con esos tiempos es casi imposible sacar el producto a tiempo, y cada 3 semanas hacer un despliegue a producción. Con microservicios sí, debido a que se pueden modificar y publicar un microservicio tras otro y al final se tiene siempre una aplicación valida.

Primera variable Time to Market

Segunda variable: secuencia simple.

Leonardo DaVinci dijo una frase bien interesante: “La simplicidad no es más que complejidad bien administrada”. El hecho de que con los microservicios se logre publicar cada 2 o 3 semanas no hace simple a la aplicación, ya sea la aplicación monolítica o distribuida implementando microservicios, no va a bajar su nivel de complejidad.

De hecho, en una aplicación monolítica, en un 99% de los casos es más probable que tenga un mejor performance que una aplicación distribuida, pero no necesariamente porque la arquitectura monolítica tenga en general un mejor performance que una distribuida.

Yo analizaría primero la capacidad del equipo de desarrollo

Por ejemplo, Microsoft Azure acaba de publicar una tecnología para manejar microservicios, también creo que Google tiene una tecnología similar, el hecho de que se publique este tipo de aplicaciones, es que esta es una tecnología o arquitectura que hasta ahora se está utilizando, por lo tanto, el buscar developers que sepan de microservicios y trabajen con microservicios son menos que en el resto de las tecnologías.

Por ende, desde el punto de vista del performance, analizaremos esto desde la perspectiva de la capacidad técnica del equipo de desarrollo que está detrás de la arquitectura.

*Entrevistador:*

Entonces, lo que usted está planteando es algo similar a lo que pasa con el RPG 400, que son sistemas muy seguros, muchos bancos los han utilizado, pero que cada vez son menos los desarrolladores

que conocen y manejan esta tecnología, lo que ha obligado a las empresas a migrar esos sistemas, es algo similar, pero a la inversa, como es algo nuevo, hay pocos desarrolladores que sepan y manejen el tema.

*Entrevistado:*

Otro punto a tener en cuenta entre arquitectura distribuida y monolítica, la monolítica me obliga a tener un Core muy complejo, la distribuida me permite tener muchas partes muy complejas que de alguna manera están sincronizadas, el truco está en que el arquitecto pueda hacer que toda esta complejidad de servicios se haga simple, desde esta perspectiva es un poco más sencillo lograr esta simplicidad, porque en teoría se monta un sólo Core y ya.

Pero les pregunto: ¿Alguna vez han recibido el código fuente de alguien más? y de casualidad han pensado "hay que volverla a hacer", yo creo que siempre, todo developer o la gran mayoría al toparse con esta situación, lo primero que dice es: "esto lo toco alguien más". Un arquitecto tiene una ventaja, conoce el negocio (o se espera que lo conozca) un developer cada cuanto cambia de empleo, cada 2 años tal vez.

Me atrevería a decir que en el mercado o al menos lo que yo he revisado, el cambiar un developer sugiere una curva de aprendizaje de más de 6 meses, y si usted lo ve desde un punto de vista técnico, le quita todo lo que tenga que ver con el negocio, lo ve como un developer.

Sabiendo que le va a entregar un código fuente y existe el 99% de probabilidades que le diga que eso hay que volverlo a hacer, ¿que preferiría entregarle?, un código fuente monolítico sumamente grande, o un montón de pedacitos tolerantes a fallos que, si a él se le ocurre la brillante idea de volver a hacer uno de ellos, aunque falle, el sistema no se va a caer.

*Entrevistado:*

En teoría el código fuente debería de estar bien escrito para que sea entendible por otro, y allí hago la siguiente reflexión: "si el señor que ha gastado varios años en aprender a codificar no es capaz de escribir el código claro y conciso para lo cual ha entrenado un muchos años, usted puede imaginarse como le quedara un documento para lo cual él no ha estudiado".

De los 100 developer que podría conocer, a lo sumo uno o dos sabe que es un diagrama de clase y menos que es un diagrama de secuencia o de flujo de datos, todos sabrán hacer un diagrama de clases o talvez un entidad relación, pero aun así UML nos dice que estos diagramas son sólo apoyos visuales, no documentación explicita, por más purista que se haga el diagrama de secuencia o de aplicación, no se

podrá explicar bien el código, hay que utilizar varios tipos de diagramas, como dicen en El Salvador, para gustos los colores.

Retomando el tema de los microservicios, hay una considerable distancia entre desarrollar cada tres semanas, hacer Scrum y ser agiles, hay una gran distancia entre dividir la aplicación en varias aplicaciones o hacer la aplicación en microservicios. Si en una aplicación de microservicios no es posible tomar un servicio particular y reemplazarlo por otro sin que su aplicación no se caiga, no es una aplicación de microservicios.

El principio básico, es construir la aplicación de tal manera que esta pueda vivir sin uno de los servicios, eso quiere decir que cada servicio que se cree, la plataforma tiene que saber cómo responder si este microservicio no está disponible.

Ahora cuantos desarrolladores conocen que se tomen el tiempo de analizar el código que están escribiendo, ¿qué pasa con ese código? ¿Y a cuantos conocen que llegan a las 8:00 am a escribir código? yo de estos conozco varios, del primer tipo conozco pocos.

*Entrevistador:*

Yo creo que eso en parte se debe a la preconcepción de lo que debe ser una carrera de licenciatura o ingeniería en sistemas informáticos, en consenso se piensa que un ingeniero o licenciado en esta carrera debería de conocer un lenguaje y hacer programas para con él.

*Entrevistador:*

Estoy completamente de acuerdo, sólo que quizá el problema no es sólo de visión, el problema es de nosotros mismos, a mis desarrolladores les he rogado durante años que se lean un libro de microservicio y ¿cuántos creen que lo han llegado a hacer?, eso no es culpa de los profesores que tuvieron, se lo puedo asegurar.

Para poder hacer microservicios y en general este tipo de arquitectura se requiere tener muchos conocimientos y mucho énfasis en querer hacerlo bien. Yo tuve una experiencia con unos desarrolladores que decían que manejaban Scrum y me dijeron que harían una presentación de cómo hacerlo, me mostraron sólo un par de pasos, análisis, desarrollo, pruebas, publicación, etc. Pero ellos no podían sacar más de un entregable por sprint, no podían hacer más de una publicación cada 2 o 3 semanas, no podían en su modelo, desarrollar y probar al mismo tiempo, en Scrum usted tiene que entregar valor al menos una vez, en un periodo de tiempo relativamente corto y se recomienda de 2 a 3 semanas.

Estos desarrolladores de entrada estaban rompiendo la norma, lo mismo pasa con los microservicios, la gente cree que un microservicio es una cosita que hace algo y ya. Muchos piensan que

por leer un libro o un blog ya pueden hacer microservicios, esto hay que estudiarlo, hay que mirarlo, analizarlo, iterarlo. También es irreal pensar que uno va a llegar al estado más purista de microservicios.

Con los microservicios hay que tener cuidado, si se mantiene simple la complejidad de la aplicación distribuida en microservicios, el desarrollar algo nuevo es relativamente fácil. Si usted tiene una aplicación monolítica, tiene que ver todo el código porque no sabe si lo nuevo que se desarrolla puede dañar algo, lo más probable cuando se le da un código monolítico a alguien para desarrollar algo nuevo, es que ocasione un bug, tras bug, tras bug.

Si la aplicación la tiene con microservicios en teoría, sólo en teoría, usted puede simplificar ese proceso de desarrollo y pruebas, y no va a tener tantos bugs, adicional a que usted puede entregar pruebas unitarias para cada microservicio.

Ahora, ¿porque utilizar monolítico? sencillo, porque casi nadie sabe hacer microservicios, por esa razón, por ejemplo, con mis desarrolladores, tengo que entregar algo funcional para dentro de 3 semanas, entonces hagámoslo monolítico, ¿por qué? Porque no tienen la experiencia, no tienen los conocimientos suficientes para sacar una aplicación con las bondades de los microservicios. Mejor malo conocido, que bueno por conocer.

Si se mete a modificar una aplicación critica implementar microservicios con un equipo que no está preparado, o esta medianamente preparado para implementar microservicios, al final del día usted va a quedar con un desastre y todos culparan a los microservicios, pero en realidad la culpa es del equipo que no lee y se entrena para implementar esa arquitectura.

*Entrevistador:*

Eso es bastante importante, porque nosotros queremos contextualizar esto, y determinar qué tan factible es utilizar microservicios acá en El Salvador

*Entrevistado:*

Es muy factible, el problema es quien va a tomarse la tarea de decirles a sus colaboradores que tienen que leer y entrenarse. Aunque en mi experiencia si usted me pregunta que tomar monolítico o microservicio, yo le diría monolítico, porque no hay experiencia para tomar un proyecto de microservicios, las universidades y el medio está plagado de ingenieros que pueden hacer cosas que funciones, que funcionen bien, pues allí hay una montañita de incertidumbre.

Pero si usted tiene un equipo de desarrolladores de confianza y una aplicación, puede ir experimentando con la aplicación, cambiando las cosas poco a poco. Cuando usted empezó a aprender a caminar hasta el día de hoy, le aseguro que se ha caído muchas veces, pero la practica hizo que

aprendiera, así se podrían abordar los microservicios, el problema es que todos a la primera caída ya nos damos por fracasados.

Entonces, ¿cuándo seria monolítico y cuando microservicios?, pues use microservicios cuando usted tenga un buen equipo, experto en microservicios, pero hacer el ejercicio y decidir pasar un monolítico a microservicios, y esperar que en las primeras 2 semanas tenga un entregable, eso es mucho pedir. Un desarrollador promedio, casi siempre llega con un folder de certificaciones, yo normalmente, ni les pregunto cuántas certificaciones tienen, mejor pregúnteles cuantos libros leen al mes, porque usted va a necesitar gente que lea y aprenda de microservicios, de domain driven design, de patrones de diseño, etc y todas estas cosas tienen algo en común, ninguna casa de software, ningún vendor, etc., las ha vuelto certificaciones, nadie le va a traer una certificación " yo soy certificado en DDD". Usted eso es lo que necesita; hace 6 años lo que tocaba aprender era SOA, hoy son microservicios.

Otro factor es la capacidad de entrenamiento del equipo, y que tanta capacidad de adaptabilidad tiene, hay un dicho que dice "las especies que sobreviven, no son las más fuertes, son las que mejor se adaptan", no necesita un equipo fuerte con cientos de certificaciones, se necesita un equipo que sea capaz de adaptarse al chip que todo se hará con microservicios, sino tiene un equipo así, ni intente hacer una aplicación de microservicios, va a quedar peor que como la tenía, ya no va a tener que leer una clase de 1000 líneas, sino 1000 clases de una línea, pero tocara leerlas todas para entender la complejidad.

En la mayoría de los casos, alguien en su zona de confort se iría por una arquitectura monolítica, por eso sólo 3 o 4 equipos tienen éxito, porque sólo estos 3 o 4 equipos entrenaron para hacer aplicaciones como Netflix, Airbnb, Amazon, etc., y se tomaron el trabajo de hacerlo y hacerlo bien.

Todas esas variables, influyen, si comparas teóricamente las arquitecturas microservicios contra monolítico, veras que microservicios ofrece una gran ventaja, pero a la hora de hacerlo tiene que comprometerse a que se necesita una gran cantidad de conocimientos y experiencia, y habrá un largo camino por recorrer que posiblemente no se tenga que recorrer, por eso hay tantas aplicaciones monolíticas y tan pocas aplicaciones de microservicios

*Entrevistador:*

Eso es muy cierto, se necesita gran cantidad de experiencia para hacer microservicios, en la lectura que hemos hecho, hemos visto que hay una gran similitud entre DDD y microservicios, es decir DDD es un medio para llegar a los microservicios, pero DDD no es algo que la mayoría de los desarrolladores conozca. En la teoría, DDD nos ayuda a descomponer dominios complejos y llegar a un nivel de desacoplamiento donde tengamos bastantes subdominios que pueden ser llevados a microservicios si se hace un buen análisis.

Entonces lo que proponemos es usar DDD para descomponer una lógica compleja que se desacople y después de ese análisis, transformarlo en microservicios para dar una solución real.

*Entrevistado:*

Bueno, DDD nos ayuda a segregar el dominio de una aplicación, que es la lógica funcional de un sistema. Hay varios componentes o cuando uno lo ve en diferentes capas se representa relativamente sencillo lo que es DDD, está la capa de interfaz gráfica, la capa de persistencia que guarda los datos, la capa da acceso a datos y arriba esta la capa de lógica de negocio y una de estas capas es una capa donde está el código para orquestar el código funcional.

Entonces, los eventos que ocurren cuando le da clic al botón guardar, le dice guárdeme estos datos (estos datos pertenecen a una persona) y al retornarlo, retorne la información de una persona, es sólo una lógica que orquesta eventos o la transformación de los datos.

¿Como llevar una arquitectura monolítica a una de microservicios?, una solución se llama Domain Driver Design, pero son principios y recomendaciones, pero no da una fórmula matemática que uno pueda aplicar, Lamentablemente, la mayoría de ingenieros buscan eso, una fórmula perfecta, creen que la metodología es una fórmula perfecta para pasar del punto A al punto B, entonces, como llego de una aplicación monolítica a una de microservicios, bueno lo primero es hacerle un análisis DDD para separar esa lógica en partes pequeñas que puedan vivir la una sin la otra, allí esta lo difícil, la forma de llegar de una arquitectura monolítica a una de microservicios, es algo que a la mayoría de ingenieros no les cuadra en su chip matemático, es decir, ¿cómo llego al dominio y subdominio más perfecto, como lo depuro?, pues probando y fallando, es decir iterando cada subdominio para ver si se puede seguir descomponiendo o si hay algo más involucrado y lo que parecía una sola acción o función en realidad se convierten en dos o tres. Los microservicios no necesariamente se desarrollan después de hacer DDD, se van construyendo en el camino.

*Conclusiones:*

Una de las principales razones de llevar una aplicación monolítica a microservicios o implementar la arquitectura de microservicios en el desarrollo de una aplicación es “Time To Market”, es decir el tiempo de reacción que el equipo de desarrollo, y por ende el negocio pueda tener para adaptar la aplicación que implementa estos microservicios para adaptarse a las necesidades del mercado, hoy por hoy los equipos (no sólo desarrollo de aplicaciones, sino del negocio mismo) tienen que tener la

capacidad de reaccionar oportunamente a las condiciones cambiantes del mercado, o ante una oportunidad de mejora que brinde una ventaja competitiva para la empresa.

La premisa Time To Market se integra no sólo con la capacidad de reacción de los equipos, sino también con filosofías de planeación como por las que propone por ejemplo COBIT5, la implementación de este tipo de arquitecturas supone una gran ventaja para las planeaciones estratégicas de la empresa que las implemente.

Enfocándonos sólo en el proceso de desarrollo de software, podemos hablar del proceso de control de cambios, si implementamos una arquitectura de microservicios donde en teoría cada microservicio se encarga de una unidad funcional única dentro del conjunto de funcionalidades involucradas en el dominio de la aplicación, un cambio en este microservicio sólo sugeriría pruebas unitarias para ese modulo en particular, también por tratarse de un sólo servicio, este sólo debería de afectar a un sólo organismo en el flujo de negocio, por lo que el proceso de control de cambios se simplifica, al contrario de una aplicación con arquitectura monolítica, no hay necesidad de probar toda la aplicación, sólo el microservicio donde se ha aplica cado el cambio, y como este opera con cierta autonomía al resto de la aplicación, no deben, las modificaciones que se hagan a este, afectar la funcionalidad de los demás microservicios (esto siempre y cuando las interfaces se respeten, es decir la forma como estos se comunican).

En nuestro medio, la mayor cantidad de implementaciones de arquitecturas monolíticas u otro tipo de aplicaciones distribuidas no se debe a que estas tengan ventajas funcionales sobre una arquitectura de microservicios, ya vimos que no, con microservicios por ejemplo, se puede con relativa facilidad implementar lenguajes de BMP para aumentar la eficiencia y eficacia en los procesos de cambio del negocio, podemos decir que se debe más bien a la capacidad de los equipos de desarrollo que actualmente existen, hay mucha más gente versada en la arquitectura monolítica u otro tipo de arquitecturas distribuidas, como MVC o cliente/servidor, que gente que realmente esté preparada para hacer frente a un proyecto de desarrollo e implementación de una aplicación que tenga una arquitectura monolítica, sin hablar del mantenimiento (preventivo/correctivo) y testeo de este. Por esa razón es más factible utilizar una alternativa a una arquitectura de microservicios en nuestro medio, al menos que se tenga el tiempo suficiente y un equipo suficientemente versátil, y con suficiente experiencia para ahondar en el tema, de lo contrario lo más probable es que este tipo de proyectos con arquitectura de microservicios fracase.

Finalmente, una forma quizá la mejor forma de lograr el diseño de una aplicación orientada a utilizar microservicios es mediante el uso de DDD, que nos permita descomponer el dominio en subdominios que representen unidades funcionales lo suficientemente simples, que permitan representar la complejidad de la lógica de la aplicación, pero hay que tener en cuenta que esta concepción muy

usualmente será iterativa, es decir, no encontraremos cada subdominio en la primera iteración, lo más probable es que siempre encontremos nuevas formas de descomponer nuestros subdominios. En ocasiones pudiéramos encontrar nuevas consideraciones funcionales o de negocio que no se tomaron en cuenta al inicio, debido a esto es importante la integración del personal del negocio.

**Experto 2**

**Nombre de Experto: Marvin Campos.**

**Nacionalidad: salvadoreño.**

**Área de campo: Arquitecto empresarial.**

*Entrevistador:*

¿Por qué se consideraría pasar de una arquitectura de monolítico a una distribuida?

*Entrevistado:*

Microservicios es un patrón arquitectónico, hay que entender primeramente el tamaño del microservicio y su respectivo alcance. Cada quien define microservicios, el deber ser es algo pequeño, simple, que encapsula una única funcionalidad, y poder tener varios microservicios que se comunican entre sí por un protocolo definido.

No es hablar de tecnología, se debería empezar primeramente en el factor humano. Hay dos cosas fundamentales en la gente cuando empiezas el tema de microservicios: Uno, el cambio de forma de pensar. Toda nuestra formación en la universidad fue lineal y no estructurada; a pesar de que las vistas estén claras como vista, lógica y modelo. Además, que las personas tengan claro los conceptos. Para muchos microservicios es una API, es algo pequeño, y ese es un problema, que todos tengan un concepto diferente del tema, y eso impide o da obstáculos al momento de arrancar con esta implementación.

Dos, entender el tamaño y el alcance de microservicios. domain driven design, es un conjunto de técnicas que permiten diseñar pensando en el dominio, para que la tecnología gire alrededor de este dominio. Este es un tema en el cual, en el papel todo corre y se mira bien chévere.

Primeramente, debemos entender ¿qué es el dominio? Es un conjunto de conocimientos, sobre el cual hay unas actividades y unos actores que realizan dichas actividades. Por lo tanto, DDD es más que modelar las interacciones de esos dominios.

¿Modelar? Es colocar a un nivel de abstracción, tal que cada uno que trabaja en el proyecto o en el problema, entiendan cuales son los elementos fundamentales de un dominio, las interacciones de cada uno de esos elementos, y representarlos en un lenguaje o esquema que los diferentes actores entiendan, y en DDD se conoce como lenguaje ubicuo, el cual es una definición bien etérea. ¿Qué es un lenguaje ubicuo? Es algo q podamos entender, pero hay varios niveles, como los q ocupas en el negocio, en tu equipo, en arquitectura, etc.

¿Qué haría yo para poder realizar esta migración de arquitecturas? Tener todo claro de lo que estamos hablando. Fundamentalmente, sin eso pasa semanas tratando de modelar, diseñar y entender, cada uno entiende de forma diferente, y no coincide nada.

Establecer un marco común: lo que tengo monolítico, modelarlo, usando DDD. Entender, cuáles son los actores, cual es el dominio principal, cual es el Core Domain, cuáles son los subdominios, y a partir de ahí entender si yo tengo, un Bounded Context (termino DDD) para cada uno de esos subdominios, que todos estén de acuerdo con ese modelo, que se haya diseñado.

Recomendación, no empezar a diseñar todos los procesos como microservicios, sino que empezar con pocos. En la teoría según el concepto de microservicios, como cada uno es totalmente independiente, se podría repartir el desarrollo de cada uno con cada desarrollador, y así trabajar de forma paralela cada uno; pero eso no pasa, y es bien difícil, siempre hay dependencias, pueden depender de una estructura de base de datos, de un servicio en común. Por eso es importante, partir de algo que no es tan grande como un monolítico llevado a un back-end, o llevarlo a un servicio que haga algo bien puntual.

El criterio para partir es empezar con un servicio que tenga un dominio, si en el camino veo que hay una parte de ese servicio que evoluciona más rápido que el resto, eso me está dando una pista que ese servicio debo partirlo, y es mejor así debido a que es mejor darse cuenta en ese momento, que después ver que el tiempo avance, y ver que tienes dos microservicios que hacen lo mismo, y lo que hemos aumentando sólo es el nivel de complejidad de la solución.

Trabajar con microservicios demanda un montón de comunicación con el equipo y la gente del proyecto. Los cambios que suceden en los contratos de intercambio de información son bien fundamentales. Cuando se arranca con microservicios, aunque hayas arrancado bien, y hayas diseñado tus modelos, y todos los entiendan, todos estén aceptos, y todos sepamos lo que estamos trabajando, venir y aterrizar eso en código no es tan fácil.

Todos deben estar conscientes de la complejidad que representa esta migración. El trabajo necesario para darle mantenimiento a los microservicios es bien grande. Si tu no montas un motor de integración continua, o despliegue continuo es un problema. Es decir, entender como son mis esquemas de despliegue, si es que una persona se encarga de subir y desplegar todo, eso no me funcionara para este tema de microservicios, necesito que se realicen de forma rápida, ¿por qué razón?, porque el desarrollo va evolucionando.

*Entrevistador:*

¿En base a tu criterio, es necesario entender y modelar antes de desarrollar? Es decir, ¿desarrollar e inventarte en el momento el microservicio es pérdida de tiempo? Si empiezas a ver tu código, en este sprint y creas un microservicio, sin modelar, ¿es una pérdida de tiempo?

*Entrevistado:*

Depende, puede que termines al final con un buen microservicio, porque si estás trabajando con DDD, si fuera una pérdida de tiempo. Algunos piensan que los microservicios, deben realizarlo sin pensarlo tanto, pero primeramente según Marvin Campos, se debe entender que se hará y cuáles son mis dominios, previamente a meter mano en el código.

*Entrevistador:*

¿Ah qué grado piensas que es necesario para que un equipo de desarrollo conocer de buenas prácticas, patrones de diseño, etc., al momento de realizar microservicios?

*Entrevistado:*

Todo esto (microservicios y DDD) requiere tiempo, practica, y no se realiza en una semana, no se cubre con una simple capacitación. El skill técnico de los desarrolladores, debe ser alto. Las buenas prácticas y patrones de diseño deben ser dominados totalmente, talvez no todos, pero los q usaras en ese desarrollo deben tenerlos claro, por lo tanto, debe ser fundamental para hacer microservicios de verdad. Entre más experto sea el equipo, más rápido podés llegar al objetivo, y el objetivo es poder evolucionar casi de manera natural a partir de tu dominio, cuantos microservicios hacer, etc.

*Entrevistador:*

¿De dónde viene la necesidad de pasar de monolito a microservicios? ¿Siempre debe aplicarse?

*Entrevistado:*

No siempre. Es bien complicado realizar un checklist de cuando sí o no hacerlo, porque dependerá del problema. Hay compañías que están en la nube, pero conciben la nube como un data center más resguardado, no me preocupo por ello, lo ven como una manera de despliegue. La nube es una forma de cómputo, y el concepto monolito no es compatible con la nube, porque todas las ventajas de las nubes se pierden con un monolito, esta es una razón.

Otra razón, tengo una empresa que va evolucionando con el tiempo, si tengo un dominio dentro de la empresa, y cada dominio evoluciona de forma diferente, no se puede pretender meter todo eso en un

monolito, y la arquitectura no me permitirá evolucionar a un buen ritmo. Si la empresa requiere más rapidez en cuanto a desarrollo, es una buena señal para partir mi monolito en microservicios.

Adicionalmente, hemos pensado únicamente como arquitectos tecnológicos, pero también tenemos que pensar en el presupuesto de la empresa. Nosotros estamos obligados a hablar con el equipo técnico, pero también con el equipo financiero, de negocio, etc. De hecho, nuestro principal aporte es que nuestro mensaje se entienda a las diferentes áreas de la empresa. Cuando piensas en un monolito, es fácil hacer un caso de negocio, que agarraremos la aplicación y la montaremos en la nube en un IaaS. Mantener un monolítico, te lleva a montar un servidor IaaS y a montar una licencia de SQL Server, si necesitas Always-On, se suma al monto. Cada una de estas acciones, tiene un costo, sin importar si contrato un PaaS.

*Entrevistador:*

¿Antes de migrar deberíamos hacer un análisis de presupuesto?

*Entrevistado:*

Totalmente, OPEX (Costo operativo). Cuando se piensa en una arquitectura de microservicios, tenemos que estar consciente que el crecimiento o la evolución del mismo impactaran mi OPEX. Si necesitamos incrementar un servidor, implica hablarle al proveedor, q lo configure, que lo agregue al cluster, pagar el servidor como tal. Todo esto va a CAPEX. En la mayoría de los casos, el OPEX para cloud es mucho menor que para on-premise.

*Entrevistador:*

¿Cuál es tu valoración de cómo realizar un análisis de dominio, desde el punto de vista, que cada subdominio puede representar uno o N microservicios?

*Entrevistado:*

Dependería del problema al que este enfrentado. Si estoy enfrentado a un problema que requiere un análisis matemático, buscaría a una persona que sepa más de matemática que yo, para q me ayude a crear un modelo matemático que pueda pasar a un sistema. En pocas palabras, hacerme acompañar de una persona que sea totalmente experta en ese dominio, fundamental para trabajar con DDD. No hablamos de un product owner, o las personas que escribe requerimientos, sino una persona que domina en gran manera el dominio.

Tener sentado al experto del dominio, junto con el equipo técnico, trabajando con BPM para que se entienda el lenguaje entre todos. Luego sentarte a modelar con la persona que entiende ese dominio. Diagrama de arquitectura, o diagrama de caja, diagrama de clases, conforme se va bajando el nivel de detalle.

*Entrevistador:*

¿El modelado de DDD es una actividad que sólo se debería hacer una vez o podría modelar mi dominio, y luego volver a hacer modelado, etc.?

*Entrevistado:*

Si se estas realizando waterfall no ocupes DDD. Todo modelo se basa en asunciones, si ese modelo cambia con el tiempo, ese modelo ya no servirá. Y los modelos cambian con el tiempo, lo que ahora es así, mañana no lo es. Puede que lo que se trabaje por un año, eso ya no representa nada para el negocio, y toque cambiarlo. Antes de hacer microservicios, se debe trabajar con Agile, tanto en el proceso de escribir requerimientos, definiciones, diseño, etc.

**Experto 3**

**Nombre de Experto: Pablo Portillo**

**Nacionalidad: salvadoreño.**

**Área de campo: Consultor**

*Entrevistador:*

¿Qué es para ti un microservicio? ¿En qué consiste?

*Entrevistado:*

A nivel técnico, el conocimiento que yo ya he aplicado de un microservicio tiene que ver con una funcionalidad o un proceso que se pueda aislar, que pueda ser reutilizable, que sea auto mantenible, ósea que no tenga dependencias con otros microservicios, es la unidad de medida más pequeña que puede tener quizá un proceso; es lo que entiendo como un servicio y lo que he aplicado como consultor. Esto va orientado a arquitectura SOA, y es donde la gente tiene deficiencias o mal entendimiento, y es que a nivel de SOA se puede tener servicios categorizados como compuestos u otros servicios que pueden tener una

funcionalidad más aterrizada, entonces un microservicio para mí es un componente que puede ser reutilizable en el futuro y que tiene una sola funcionalidad, y que puede ser orquestado desde un servicio un poco más compuesto.

*Entrevistador:*

¿Por qué considerarías necesaria la migración de una arquitectura monolítica a una arquitectura distribuida y bajo qué criterios?

*Entrevistado:*

Primero te voy a hablar de la experiencia que he tenido, pero también a nivel conceptual de lo que para mí debería tomarse en cuenta al decidir dividir una aplicación monolítica en microservicios. En primer lugar, que el esfuerzo hay desmenuzar una aplicación monolítica a microservicios, se compensa a nivel de reusabilidad de los microservicios, si a nivel de desarrollo voy a minimizar el tiempo en el que voy a desarrollar funcionalidades a partir de la reusabilidad de los microservicios, desde ahí estoy ganando un punto grandísimo; eso es importante, al dividir la aplicación en servicios estoy ganando reusabilidad o no.

Segundo, ¿será que la aplicación monolítica está teniendo problemas de rendimiento o de cuellos de botella? Esa sería la otra pregunta, y eso me viene a la mente cuando trabaje para una empresa como consultor, había una aplicación que no era capaz que no respondía en la velocidad que ellos solicitaban, porque toda la funcionalidad estaba escrita en el monolítico, sin importar que ocupaba uno de los frameworks más rápidos de Java como lo es Spring, pero por la forma en que estaba construida y con muchas capas brindaba esa lentitud en su procesamiento, pero al desmenuzar la aplicación en microservicios, se mejoraron los tiempos de respuesta casi en un 70%, un proceso veloz a través de esta arquitectura.

En conclusión, verificar si ganare reusabilidad, y si la aplicación monolítica está teniendo problemas de performance, como justificación para esta migración.

*Entrevistador:*

¿Es para ti mala una arquitectura monolítica?

*Entrevistado:*

Yo pensaría que no, más que todo depende de la naturaleza del contexto de la aplicación monolítica, porque si estas en una empresa pequeña donde tienes dos desarrolladores, y tienes procesos

que no reutilizaras, desde mi punto de vista yo no perdería el tiempo en tratar de desmenuzar la aplicación monolítica en microservicios.

*Entrevistador:*

¿Cuál sería el punto de partida para llevar a cabo esta migración de arquitecturas?

*Entrevistado:*

Yo creo que lo importante ahí es el rol del arquitecto con la suficiente experiencia de poder tomar una aplicación monolítica y comenzar a generar los flujos de la aplicación. Por ejemplo, con BPM y sabes que es una aplicación que tiene que ver con el manejo de GPS, entonces desmenuzar los procesos que más se utilizar en todo el core de la aplicación, es decir, si el 50% de los llamados que se hacen a la aplicación es para obtener reportes, y los reportes se construyen a través de consultas a una tabla que contienen tramas y esas tramas se proyectan en un plano cartesiano, etc. entonces esos flujos son los que se tienen que empezar a identificar y de esa manera, identificar esos flujos que se repiten y de esa manera obtener los microservicios a nivel de arquitectura que se deberían de proponer para iniciar la documentación, modelado, etc.

*Entrevistador:*

¿Será necesario entender y modelar antes de desarrollar, o viceversa?

*Entrevistado:*

Fíjate que yo ya he trabajado en ese sentido con personas que justamente hacen primeramente el modelado y luego a desarrollar, y desde mi punto de vista eso es ganancia para el desarrollador, porque el desarrollador ya sabe que es lo que va a realizar, ya sabe justamente lo que va a utilizar para realizar su trabajo. Mientras si nos vamos a un punto de vista más tradicional, donde el desarrollador se sienta y en el momento que desarrolla empieza a inventar como va a solucionar la situación, es ahí donde las aplicaciones no se estandarizan, cada programador trabaja de la forma que le gusta más. En conclusión, hay mucha ganancia sentarse primero a modelar y luego desarrollar dicha solución.

*Entrevistador:*

¿Será que previamente a enfrentarse a una migración de esta magnitud, habría que sentarse a realizar un análisis de presupuesto para esta implementación? ¿Tiene alto costo financiero la implementación de esta nueva arquitectura de microservicios?

*Entrevistado:*

Fíjate que esos temas son bien delicados, y son una de las cosas que uno como arquitecto tiene que saber vender la idea. Al final, las personas que aprueban los proyectos no son las personas que saben de tecnología o que puedan ver un beneficio tecnológico, sino más bien como un tema económico, es decir, ROI (Retorno de la Inversión). En nuestro caso, nosotros nos vimos en el problema que a pesar de que la aplicación sea monolítica, tenía muchos clientes y los programadores eran escasos, y como podríamos darle mantenimiento a dicha aplicación, porque sólo por la migración no íbamos a dejar a los clientes sin ocupar la aplicación; hay que saber encontrar la verdadera justificación, y así indicar que esta migración proveería un valor a la empresa. Adicionalmente, como podemos hacer tangible ese beneficio que nosotros queremos llevarle a la empresa, esta migración a una arquitectura de microservicio. También podría ser que digamos, este microservicio nos costará 20 horas hombre, y que se traducen en 20 por el monto total que se le paga al desarrollador, pero eso nos permitirá que en otro desarrollo futuro nosotros no tengamos que codificar nuevamente, y esto al final lo podés medir y dices si es un ROI de lo que hicimos, estamos siendo más rápidos al construir aplicaciones. Pero como te digo, la parte de análisis de los costos de esta migración, deberíamos acompañarlo con el ROI, indicar en cuanto tiempo podemos garantizar que lo que estamos haciendo a través de la reusabilidad, va a brindar un beneficio a la empresa.

*Entrevistador:*

¿Habrá un sólo camino o algo escrito para realizar este tipo de migraciones hacia una arquitectura de microservicios?

*Entrevistado:*

Fíjate que como casi todas las cosas en tecnología y como experiencia que tengo en mi día laboral, ninguna realidad específica o una solución especifica que se aplique a las empresas no existe. Este tipo de migración tiene que estar relacionada o específica a la empresa, a la realidad de la empresa, a la cultura de la empresa, a las posibilidades económicas de la empresa, a la disponibilidad de recurso humano que tenga la empresa. Si bien, tú puedes definir un lineamiento genérico, a nivel de la implementación de cada una de las fases que tú puedas definir, va a depender de la realidad de la compañía.

## Apéndice 2. Matriz de congruencia

| No. | Pregunta de Investigación | Objetivo | Metodología | Instrumentos | Variables | Indicadores |
|---|---|---|---|---|---|---|
| 1 | ¿Cómo fue el proceso de descomposición del dominio usando domain driven design? | Identificar los componentes del dominio que se migrarán a microservicios distribuidos. | 1. Leer la documentación técnica de la aplicación.<br>1.1. Requerimientos funcionales<br>1.2. Identificar arquitectura inicial de la aplicación.<br>1.3. Identificar entradas y salidas de cada componente.<br>1.4. Identificar procesos.<br><br>2. Leer documentación de modelado de dominio.<br>2.1. Identificar las capas de presentación, aplicación, dominio, infraestructura.<br>2.2. Identificar los subdominios<br>2.3. Identificar los casos de uso para cada componente subdominio. | 1. Diseño de modelado de dominio<br>2. Documento técnico de la aplicación<br>3. Arquitectura Inicial<br>4. Requerimientos Funcionales<br>5. Pruebas de usuario<br>6. Métricas de rendimiento | 1. Funcionalidad<br>2. Diagnóstico del Rendimiento | 1.1. Casos de uso de cada componente subdominio<br><br>2.1 Tiempos de respuesta.<br>2.2. Transacciones por unidad de tiempo. |
| 2 | ¿Cuál diseño de arquitectura se propone para solventar la problemática actual de la aplicación? | Definir una arquitectura distribuida usando microservicios partiendo del modelo de dominio.<br><br>Elaborar una guía sobre la migración de un sistema monolítico a un sistema distribuido. | 1. Indagar en documentación bibliográfica sobre microservicios y DDD.<br>1.1. Realizar un compendio con respecto a microservicios y DDD.<br><br>2. Validar diferentes puntos de vista mediante juicio de expertos.<br>2.1. Obtener recomendaciones y buenas prácticas para establecer una arquitectura distribuida usando DDD y microservicios. | 1. Documentación bibliográfica<br>2. Juicio de expertos<br>3. Documento de investigación de caso de estudio. | 1. Reutilización y escalabilidad de componentes de software.<br><br>3. Calidad del documento de recomendaciones y buenas prácticas. | 1.1. Nivel de reutilización de especificaciones<br>1.2. Facilidad de despliegue de cada componente de software.<br>1.3. Facilidad de modificar o actualizar componentes, sin dar de baja por completo al sistema.<br><br>2.1. Facilidad de entendimiento<br>2.2. Claridad en cuanto a lectura y escritura.<br>2.3. Confiabilidad de la información (Referencias que lo respalde) |

Tabla 3. Matriz de congruencia para caso de estudio

## Apéndice 3. Perfil de expertos

El perfil de las personas expertas que se buscó para llevar a cabo las entrevistas de este trabajo tenía que cumplir lo siguiente:

- Ingeniero/Licenciado en Sistemas Informáticos que se desempeñe como Arquitecto de Aplicaciones o Integraciones con al menos cuatro años de experiencia en el diseño de aplicaciones con una arquitectura distribuida.
- Poseer conocimientos en diferentes tecnologías para aplicaciones empresariales con énfasis en la implementación de patrones de diseño y patrones arquitectónicos de N-capas, SOA, y microservicios.
- Conocimiento sólido en domain driven design.
- Conocimiento deseable en COBIT 5 e ITIL con el propósito de gestionar los procesos empresariales e integración de procesos de IT con el negocio.
- Excelente aptitud de comunicación.