UNIVERSIDAD NACIONAL DEL ALTIPLANO

ESCUELA DE POSGRADO

DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

TESIS

MODELO DE COMPOSICIÓN DE MICROSERVICIOS PARA LA IMPLEMENTACIÓN DE UNA APLICACIÓN WEB DE COMERCIO ELECTRÓNICO UTILIZANDO KUBERNETES

PRESENTADA POR:

DONIA ALIZANDRA RUELAS ACERO

PARA OPTAR EL GRADO ACADÉMICO DE:

DOCTORIS SCIENTIAE EN CIENCIAS DE LA COMPUTACIÓN

PUNO, PERÚ

2017

UNIVERSIDAD NACIONAL DEL ALTIPLANO

ESCUELA DE POSGRADO

DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

TESIS

MODELO DE COMPOSICIÓN DE MICROSERVICIOS PARA LA IMPLEMENTACIÓN DE UNA APLICACIÓN WEB DE COMERCIO ELECTRÓNICO UTILIZANDO KUBERNETES

PRESENTADA POR:

DONIA ALIZANDRA RUELAS ACERO

PARA OPTAR EL GRADO ACADÉMICO DE:

DOCTORIS SCIENTIAE EN CIENCIAS DE LA COMPUTACIÓN

APROBADA POR EL SIGUIENTE JURADO:

PRESIDENTE

Dr. EDGAR ELOY CARPIO VARGAS

PRIMER MIEMBRO

Dr. MARCO ANTONIO QUISPE BARRA

SEGUNDO MIEMBRO

Dr. NORMÁN JESÚS BELTRÁN CASTAÑÓN

ASESOR DE TESIS

Dr. ELVIS AUGUSTO ALIAGA PAYEHUANCA

Puno, 20 de Diciembre del 2017

ÁREA: Estrategias metodológicas de educación superior
TEMA: El uso de recursos didácticos y materiales con el rendimiento académico de los estudiantes
LÍNEA: Comprobación de la eficiencia y eficacia de estrategias metodológicas en la educación superior

DEDICATORIA

Con mucho afecto para mis seres queridos: mi hijo Anthony Miguel Aceituno Ruelas, mis padres Andrés Ruelas Luque y Paulina Acero Calsin, mis hermanos Elio R. Ruelas Acero y Edinson R. Ruelas Acero (Q.E.P.D.).

AGRADECIMIENTOS

- A la Universidad Nacional del Altiplano por la formación tanto nivel de pregrado y posgrado que he recibido.
- A los miembros del jurado: Dr. Edgar Eloy Carpio Vargas, Dr. Marco Antonio Quispe Barra, Dr. Normán Jesús Beltrán Castañón y Dr. Elvis Augusto Aliaga Payehuanca; por sus sugerencias y recomendaciones para mejorar la presente investigación.

ÍNDICE GENERAL

Pág.

CAPÍTULO II

PLANTEAMIENTO DEL PROBLEMA

CAPÍTULO III

MATERIALES Y MÉTODOS

CAPÍTULO IV

RESULTADOS Y DISCUSIÓN

ÍNDICE DE TABLAS

Pág.

ÍNDICE DE FIGURAS

Pág.

ÍNDICE DE ANEXOS

Pág.

SIGLAS Y ABREVIATURAS

API: Interfaz de programación de aplicaciones.

BPMN: Modelo y notación de procesos de negocio.

DNS: Sistema de nombres de dominio.

HTTP: Protocolo de transferencia de hipertexto.

IEEE: Instituto de Ingeniería Eléctrica y Electrónica

IP: Protocolo de Internet.

JSON: Notación de objetos JavaScript.

JWT: JSON Web Token.

PVC: Demanda de volúmenes de datos persistentes.

REST: Transferencia de estado representacional.

SOA: Arquitectura orientada a servicios.

SOAP: Protocolo de acceso simple a objetos.

XML: Lenguaje marcado extensible.

URI: Identificador de recursos uniforme.

URL: Localizador uniforme de recursos.

.

RESUMEN

El comercio electrónico ha crecido en los últimos años y a consecuencia de ello ha recibido una mayor atención por parte de las empresas, las cuáles vienen invirtiendo en la implementación de aplicaciones web; incrementado su demanda y la existencia de usuarios concurrentes en determinados tiempos, ocasionando un mayor nivel de exigencia; que conlleva a problemas como la disponibilidad limitada, tiempo de respuesta excedida, y el rendimiento limitado al acceder a la aplicación web del comercio electrónico. Por otro lado la arquitectura de microservicios es una nueva tendencia que crece rápidamente en el mundo empresarial, sin embargo existe poca literatura de composición de microservicios. El objetivo de esta investigación ha sido proponer un modelo de composición de microservicios para la implementación de una aplicación Web de comercio electrónico utilizando la tecnología Kubernetes, para lo cual se utilizó el modelo y notación el proceso de negocio de comercio electrónico, el diseño arquitectónico de la composición de los microservicios, la implementación de los microservicios en forma independiente la evaluación de éstos se ha realizado a través de atributos de calidad, y la validación del modelo propuesto usando pruebas de carga. Como resultado se obtuvo que la aplicación web funciona con una mejora significativa en un 104 %, en los indicadores de rendimiento, disponibilidad y tiempo de respuesta, en comparación con una aplicación web basado en el modelo monolítico. Por lo que el modelo de composición de microservicios presentado tiene un funcionamiento significativo.

Palabras clave: Aplicación web, comercio electrónico, disponibilidad, modelo de composición de microservicios, rendimiento y tiempo de respuesta.

ABSTRACT

Electronic commerce has emerged more in recent years and as a result it has received more interest from companies, and they are investing on web application implementations; as it has been increased demand and concurrent users at certain times, resulting in a higher level of demand; which leads to issues such as limited availability, response times exceeded, and lower performance while accessing to the e-commerce web application relative to demand. Meanwhile, the microservices architecture is a new trend that is growing rapidly in the business world, but there is little literature related to microservices composition. This research's goal was to propose a microservice composition model aimed towards implementing an e-commerce web application using Kubernetes technology, for this purpose, the model and notation was used for the e-commerce business process, the architectural design of the microservices composition, independent microservices implementation, the evaluation has been done through quality attributes, and validation of this proposed model has been done using load tests. As a result, it was obtained a web application that improved significantly by 104%, referring to performance indicators, availability and response time, all that in comparison to a traditional web application based on the monolithic model. Therefore, this proposed microservice composition model has a significant performance.

Keywords: Availability, electronic commerce, microservice composition model, performance, response time and web application.

INTRODUCCIÓN

Las empresas que eran un referente en su sector, han perdido su posición dominante y han quedado relegadas debido a que no se han adaptado a los cambios tecnológicos de la era. Las empresas requieren software escalable, que se adapten al creciente número de usuarios y transacciones, debido a que sus modelos de negocio son cambiantes y por ello necesitan estar a la vanguardia de las tecnologías emergentes, como la arquitectura de microservicios que promete ventajas frente a arquitecturas monolíticas.

La presente investigación corresponde al área de ingeniería del software, en la línea de arquitectura del software con el tema de composición de microservicios. La composición de servicios permite integrar varios servicios para brindar una solución software; los microservicios es un enfoque arquitectónico reciente, sin embargo, existe una limitada literatura de composición de microservicios. En la presente investigación se tiene como objetivo, proponer un modelo de composición de microservicios para la implementación de una aplicación web de comercio electrónico, para lo cual se utilizó Kubernetes como principal herramienta tecnológica y el enfoque teórico de la arquitectura de microservicios.

La presente investigación se ha estructurado en cuatro capítulos de la siguiente manera:

En el **Capítulo I.** Revisión de la literatura: se desarrolla la revisión bibliográfica expresada en el marco teórico y los antecedentes de investigación.

En el **Capítulo II.** Planteamiento del problema: se presenta la identificación del problema de investigación, la formulación del problema de investigación, la justificación, los objetivos y las hipótesis de investigación.

En el **Capítulo III.** Materiales y métodos: se presenta los materiales tecnológicos y métodos teóricos que se utilizó en la investigación, también se describe la población y muestra que se consideraron en la investigación.

En el **Capítulo IV.** Resultados y Discusión: se exponen los resultados obtenidos del modelo de composición de microservicios para la implementación de una aplicación Web de comercio electrónico en los indicadores de tiempo de respuesta, disponibilidad y rendimiento.

CAPÍTULO I

REVISIÓN DE LITERATURA

1.1 MARCO TEÓRICO

1.1.1 Arquitectura de software

La arquitectura de software según la IEEE es definida como la organización fundamental de un sistema, incorporada en sus componentes, sus relaciones entre sí y el medio ambiente y los principios que rigen su diseño y evolución. En la literatura, existen diferentes definiciones respecto a la arquitectura de software, sin embargo la definición que se usa con mayor frecuencia, es que la arquitectura describe aspectos estructurales de un sistema software en particular, que comprende elementos de software, las propiedades externamente visibles de esos elementos y las relaciones entre ellos. (Bass, Clements, y Kazman, 2012).

Por su parte Garlan y Shaw (1993), consideran que la arquitectura del software va más allá de los algoritmos y las estructuras de datos del cómputo; el diseño y la especificación de la estructura general del sistema surge como un nuevo tipo de problema. Los problemas estructurales incluyen la organización general y la estructura de control global; protocolos para comunicación, sincronización y acceso a datos; asignación de funcionalidad a elementos de diseño; distribución física; composición de elementos de diseño; escala y rendimiento; y selección entre alternativas de diseño.

La arquitectura define los elementos que conforma la solución software, así mismo la arquitectura incorpora información sobre cómo los elementos se relacionan entre sí, esto significa que omite específicamente cierta información sobre elementos que no pertenecen a su interacción. Por lo tanto, una arquitectura es ante todo, una abstracción de un sistema, que suprime detalles de elementos que no afectan como se usa, por quienes será usado, relacionado o interacción con otros elementos. (Bass *et al*., 2012).

1.1.1.1 Creación de una arquitectura de software

Existen métodos y guías para la definición de la arquitectura, muchos de los cuales se focalizan en los requisitos funcionales, sin embargo es posible crear una arquitectura basada en las necesidades de atributos de calidad.

En la Figura 1, se muestra las etapas del proceso de modelado de la arquitectura de software, que consiste en definir los requerimientos que involucra crear un modelo, desde los requerimientos que guiarán el diseño de la arquitectura basado en los atributos de calidad esperados. El diseño de la arquitectura involucra definir la estructura y las responsabilidades de los componentes que comprenderán la arquitectura de software. La validación involucra probar la arquitectura, típicamente pasando a través del diseño contra los requerimientos actuales y cualquier posible requerimiento a futuro.

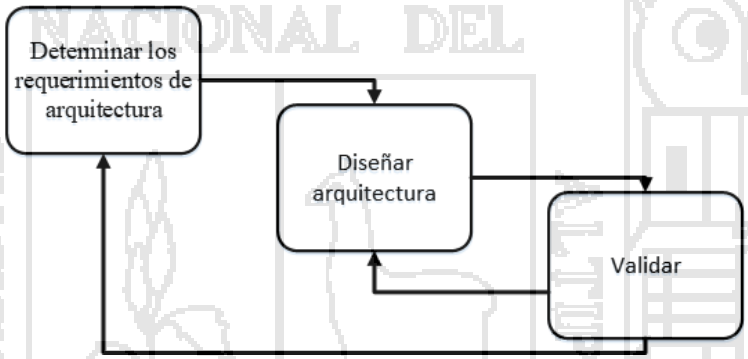


Figura 1. Proceso de modelado de arquitectura de software.
Fuente: (Bass *et al.*, 2012).

Dentro del proceso de modelado de la arquitectura de software, en la fase de validación se busca aumentar la confianza del equipo de diseño con respecto a que la arquitectura es adecuada para cumplir con los requerimientos del sistema; aunque se puede estar actuando sobre un sistema existente o nuevo, al final, el resultado del modelado es un diseño de arquitectura de software, por lo que el proceso de validación puede ser el mismo para ambos casos. Para el proceso de validación se puede emplear la técnica de prototipo, que son versiones reducidas de la aplicación deseada, creadas específicamente para poner a prueba algunos los aspectos del diseño de alto riesgo, se utilizan normalmente con dos objetivos: prueba de concepto, que busca que la arquitectura como está diseñada pueda construirse en una forma que pueda

satisfacer los requerimientos; y la prueba de tecnología que, menciona que tecnologías son seleccionadas para implementar la aplicación.

En la Figura 2, se muestra el proceso del diseño de la arquitectura del software, partiendo de los requerimientos de la arquitectura, el siguiente paso es el más crítico, que comprende escoger la arquitectura de referencia, por lo que se debe basarse en arquitecturas de referencia reconocida por la academia y tanto por la industria, y que éstas tengan una buena documentación. En la asignación de componentes se definen los principales componentes; y finalmente se diagrama las vistas de la arquitectura y la documentación respectiva.

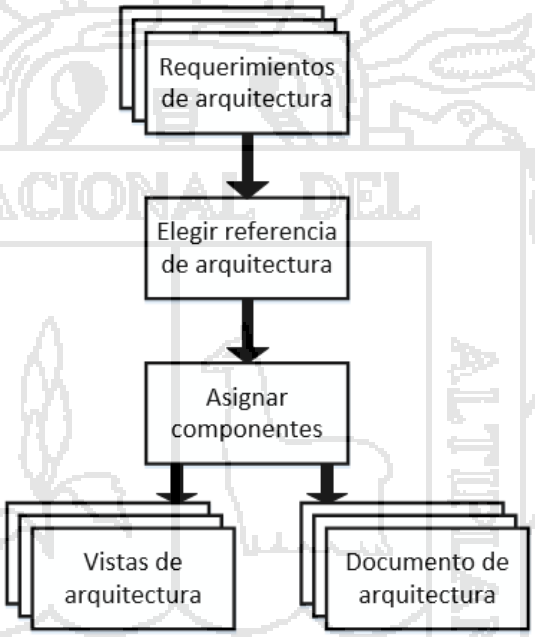


Figura 2. Diseño de la Arquitectura del Software.
Fuente: (Bass *et al.*, 2012).

1.1.1.2 Arquitectura Monolítica

Una arquitectura monolítica es el modelo tradicional unificado para el diseño de un producto de software, donde monolítico, en este contexto, significa compuesto todo de una sola pieza. El software monolítico está diseñado para ser autónomo; los componentes del programa están interconectados e interdependientes en lugar de estar débilmente acoplados, como es el caso de los programas de software modulares. En una arquitectura estrechamente acoplada, cada componente y sus componentes asociados deben estar presentes para que el código sea ejecutado o compilado. Una arquitectura monolítica

dicta que una aplicación consiste en componentes que están estrechamente vinculados entre sí y debe desarrollarse, implementarse y administrarse como una sola entidad donde la Figura 3 ilustra esta descripción, ya que todos se ejecutan como un único proceso del sistema operativo y se escala mediante la replicación de todas las funciones en servidores múltiples.(Fowler y Lewis, 2014).

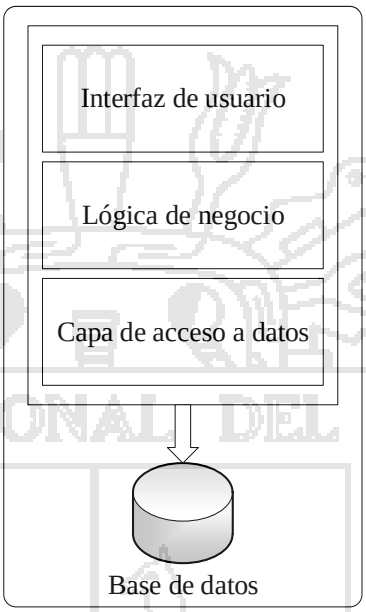


Figura 3. Arquitectura monolítica de una aplicación.
Fuente**:** (Fowler y Lewis, 2014).

Las aplicaciones monolíticas también son definidas como aquellas en las que el software se estructura en grupos funcionales acoplados, involucrando los aspectos referidos a la presentación, procesamiento y almacenamiento de la información y son implementadas dentro de un solo componente de software. (Garlan, 2000).

1.1.1.3 Arquitectura Orientada al Servicio

La Arquitectura Orientada al Servicio (SOA) es un modelo en el que la lógica de una aplicación se descompone en varias unidades más pequeñas, que conjuntamente implementan una pieza más grande de la lógica del negocio. La arquitectura orientada al servicio promueve que estas unidades individuales existen independientemente pero no aisladas entre sí, a su vez podrían ser distribuidas. Esto requiere que estas unidades operen sobre la base de ciertos principios que les permitan evolucionar independientemente, manteniendo al

mismo tiempo la uniformidad y la estandarización entre ellas. En la arquitectura orientada al servicio estas unidades lógicas se conocen como servicios. (Bellido, 2015).

Un servicio es un elemento que se comprende en términos de la utilidad y funcionalidad que brinda, por lo tanto, puede apartarse del negocio o problema para el cual debe ser útil o funcional. Un servicio es definido también como la captura de la funcionalidad con un valor de negocio que está listo para ser usado. Es provisto por servidores, para lo cual requiere de una descripción que pueda ser accedida y entendida por los clientes potenciales. Los servicios de software son servicios útiles y funcionales provistos por sistemas de software. (Erl, 2005).

Richards (2015) menciona, que la arquitectura orientada a servicios presenta varios conceptos fundamentales que son: un conjunto de servicios que la empresa desea ofrecer a sus clientes, y a otras áreas de la organización; un estilo de arquitectura orientado a servicios requiere un proveedor de servicios, mediación y un solicitante del servicio con una descripción del servicio; un conjunto de principios, modelos y criterios arquitectónicos que abordan características como modularidad, encapsulación, acoplamiento abierto, separación de elementos de interés, reutilización y composición; un modelo de programación completo con estándares, herramientas y tecnologías que admite servicios web, servicios REST y otros tipos de servicios y una solución de middleware optimizada para la coordinación, orquestación, supervisión y gestión de estos servicios.

1.1.1.4 Arquitectura de microservicios

Los microservicios son pequeños servicios autónomos que trabajan juntos. Con pequeños se refiere a realizar un servicio específico y bien realizado. Respecto a lo autónomo estos servicios deben ser capaces de cambiar independientemente uno del otro y ser desplegados por sí mismos sin necesidad de que los consumidores cambien.(Newman, 2015)

Los microservicios también son explicados de la siguiente forma, dada una aplicación éste se descompone en módulos lógicos que son completamente

autónomos e independientes y realizar su despliegue en un entorno donde se gestionan colectivamente.(Janakiram, 2017).

Los microservicios son considerados como un enfoque para desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo una API de recursos HTTP. Estos servicios se basan en capacidades empresariales y se pueden implementar de forma independiente mediante una maquinaria de despliegue totalmente automatizada. Existe un mínimo de gestión centralizada de estos servicios, que puede escribirse en diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos.(Fowler y Lewis, 2014).

La arquitectura de microservicios desarrolla una sola aplicación como un conjunto de pequeños servicios, cada uno funcionando en su propio proceso y comunicándose con mecanismos ligeros. Estos servicios se basan en capacidades empresariales y de misión, y se pueden desplegar de forma independiente mediante maquinaria totalmente automatizada.(Bakshi, 2017).

Los microservicios permiten estructurar los sistemas Software de la misma manera que estructuramos nuestros equipos, dividiendo las responsabilidades entre los miembros del equipo y asegurando que sean libres de ser propietarios de su trabajo. A medida que desenredamos éstos sistemas, cambiamos el poder de los órganos de gobierno centrales a equipos más pequeños que pueden aprovechar las oportunidades rápidamente y mantenerse ágiles porque entienden el software dentro de límites bien definidos que controlan.(Bonér, 2016).

En la Figura 4, se muestra la topología general de la arquitectura de microservicios que consta de sólo dos componentes principales: componentes de servicio y, opcionalmente, una capa API. Desde el punto de vista de la implementación, es posible que tenga otros componentes, como un componente de registro y descubrimiento de servicios, un componente de supervisión de servicio y un gestor de despliegue de servicios.(Newman, 2015).

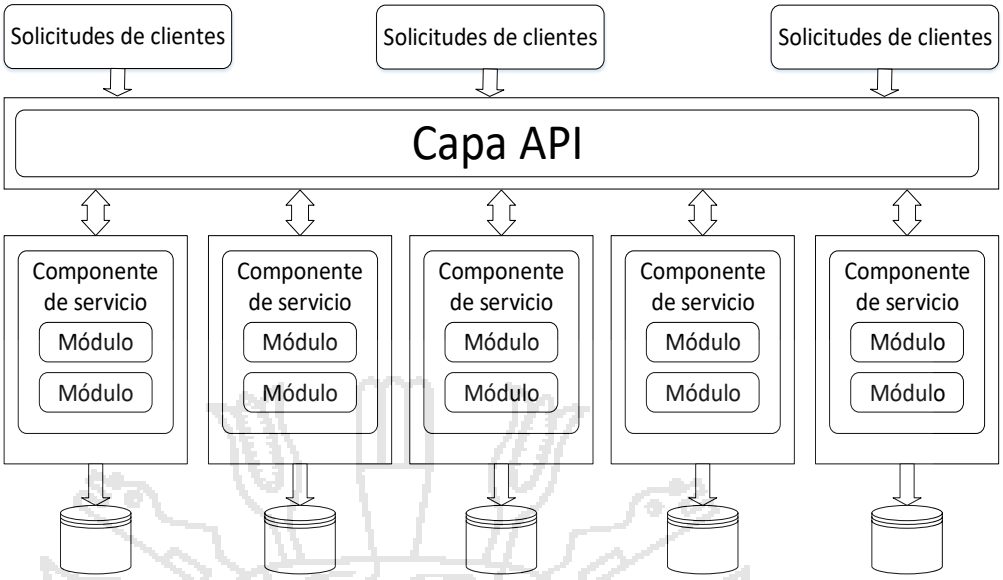


Figura 4. Topología de la arquitectura de microservicio.

Fuente: (Newman, 2015).

1.1.1.5 Características y ventajas de la arquitectura de microservicios

Se considera que no existe una definición formal del estilo arquitectónico de los microservicios, pero lo describen a través de características comunes para las arquitecturas que se ajustan a la etiqueta. Existen ciertas características comunes alrededor de la organización en torno a la capacidad empresarial, el despliegue automatizado, la inteligencia en los puntos finales y el control descentralizado de idiomas y datos. (Lewis y Fowler, 2016).

- La componetización a través de servicios es una de sus características principales, una razón principal para usar los servicios como componentes es que los servicios son desplegables de forma independiente. Si tiene una aplicación que consta de varias bibliotecas en un único proceso, un cambio en cualquier componente único tiene como resultado replegar toda la aplicación. Pero si esa aplicación se descompone en múltiples servicios, puede esperar que muchos cambios de servicio único solo requieran que el servicio sea reasignado.
- La organización alrededor de capacidades empresariales es otra característica de los microservicios que busca dividir una aplicación grande en partes, el enfoque de Microservicio para la división es diferente, dividiéndose en servicios organizados alrededor de la capacidad

empresarial. Dichos servicios requieren una implementación amplia de software para esa área de negocio, incluyendo interfaz de usuario, almacenamiento persistente y cualquier colaboración externa. En consecuencia, los equipos son inter funcionales, incluyendo toda la gama de habilidades necesarias para el desarrollo: experiencia de usuario, base de datos y gestión de proyectos.

- Productos no proyectos es una característica de los microservicios donde se ve que la mayoría de los esfuerzos de desarrollo de aplicaciones utilizan un modelo de proyecto, donde el objetivo es entregar algún software que se considera que se ha completado. Al terminar el software se entrega a una organización de mantenimiento y el equipo de proyecto que lo construyó se disuelve. Los microservicios tienden a evitar este modelo, prefiriendo en cambio la noción de que un equipo debe poseer un producto durante toda su vida útil, es decir cada equipo de desarrollo tienes asignado un microservicio para ser desarrollado. La mentalidad de producto, se vincula con el vínculo con las capacidades empresariales. En lugar de mirar el software como un conjunto de funcionalidades para ser completado, hay una relación continua donde la pregunta es cómo el software puede ayudar a sus usuarios a mejorar la capacidad empresarial.

Respecto a las ventajas de la arquitectura de microservicios se menciona que ésta impone un nivel de modularidad que en la práctica es extremadamente difícil de conseguir con una base de código monolítico, donde los servicios individuales son mucho más rápidos de desarrollar y mucho más fáciles de entender y mantener; destacan también que esta arquitectura permite que cada servicio sea desarrollado independientemente por un equipo que se centre en ese servicio. Los desarrolladores son libres de elegir cualquier tecnología. (Richardson y Smith, 2016).

La arquitectura de microservicios, para los arquitectos y desarrolladores software promete un nivel de control y velocidad sin precedentes a medida que ofrecen nuevas experiencias web innovadoras a los clientes. Así mismo como ventajas principales se mencionan a la escalabilidad en escenarios distribuidos colaborativos y las posibilidades mejoradas de desarrollo y operación de servicios.(Alpers *et al.*, 2015).

1.1.1.6 Modelamiento de microservicios

El modelo impulsado para el dominio proporciona una forma de representar el espacio del problema del mundo real para que todos los conceptos y datos importantes del mundo real permanezcan intactos en el modelo. El modelo de dominio así capturado respeta las diferencias, así como el acuerdo en los conceptos en varias partes del espacio problemático. Además, proporciona una forma de dividir el espacio problemático en particiones independientes manejables y facilita que tanto los desarrolladores como las partes interesadas se centren en el área de interés y sean ágiles. El diseño impulsado por dominio asegura que el software desarrollado cumpla con las necesidades del negocio. (Kharbuja, 2016).

Se presenta las fases básicas para implementar el diseño impulsado por el dominio:

- El lenguaje ubicuo, el lenguaje ubicuo es un lenguaje común acordado entre los expertos de dominio y los desarrolladores de un equipo. Es importante tener una comprensión común sobre los conceptos de un dominio del problema y el lenguaje ubicuo es la manera de asegurarlo. Por lo tanto, el vocabulario común y los modelos de dominio forman el núcleo del lenguaje omnipresente y la única forma de encontrar el mejor lenguaje ubicuo es mediante la aplicación extensiva de la comunicación. El lenguaje ubicuo no es solo una recopilación o documentación de términos, sino el enfoque de comunicación dentro de un equipo.

- El diseño estratégico, proporciona una forma de dividir los modelos de dominio completo en partes pequeñas, manejables e interoperables que pueden funcionar junto con una baja interdependencia para proporcionar las funcionalidades de todo el dominio. El objetivo es dividir el sistema en partes modulares que se pueden integrar fácilmente. El diseño estratégico especifica dos pasos principales que son: primero dividir el dominio del problema en subdominios, el dominio representa el espacio problema resuelto por el software. El dominio se puede dividir en varios subdominios basados en la lógica de negocio y cada subdominio es responsable de cierto proceso del negocio. segundo Identificar contextos

delimitados, la idea de contexto delimitado proporciona la aplicabilidad del lenguaje ubicuo dentro de su límite, llevando a cabo una responsabilidad específica. Fuera de los contextos delimitados, los equipos tienen diferentes idiomas ubicuos con diferentes términos, conceptos, significados y funcionalidades.

Kharbuja (2016) también recomienda el uso de caso de uso; que es una secuencia de acciones realizadas por el sistema para producir un resultado observable de valor para un usuario en particular; para el proceso de identificación y modelamiento de los microservicios.

1.1.2 Composición de servicios

La composición del servicio se originó de la necesidad de alcanzar una meta predeterminada que no puede ser realizada por un servicio independiente. Internamente, en una composición, los servicios pueden interactuar entre sí para intercambiar parámetros, por ejemplo, el resultado de un servicio podría ser el parámetro de entrada de otro servicio.(Claro *et al*., 2006).

Una de las principales fortalezas clave de las arquitecturas orientadas a servicios, es el concepto de composición de servicios para reutilizar y combinar los servicios existentes con el fin de lograr una funcionalidad nueva y superior.(Haupt *et al*., 2014).

La orquestación y coreografía de servicios son técnicas de composición, su diferencia no siempre es clara. Estos conceptos de comunicación de servicio se utilizan tanto en Microservicios como SOA.(Richards, 2015).

Las técnicas de orquestación y coreografía describen dos aspectos para crear procesos empresariales a partir de múltiples servicios. Los dos términos se superponen un poco, pero la orquestación se refiere a un proceso empresarial ejecutable que puede interactuar con los servicios Web internos y externos. La orquestación siempre representa el control desde la perspectiva de una parte. Esto lo distingue de la coreografía, que es más colaborativa y permite a cada parte implicada describir su parte en la interacción. Los estándares de orquestación y coreografía propuestos deben cumplir con varios requisitos técnicos que abordan el lenguaje para describir el flujo de trabajo del proceso y la infraestructura de apoyo.(Peltz, 2003).

1.1.2.1 Orquestación

Las composiciones de servicios construyen nuevos servicios orquestando un conjunto de servicios existentes; la orquestación de servicios se refiere a la coordinación de múltiples servicios a través de un mediador centralizado como un consumidor de servicios o un centro de integración. (Strunk, 2010).

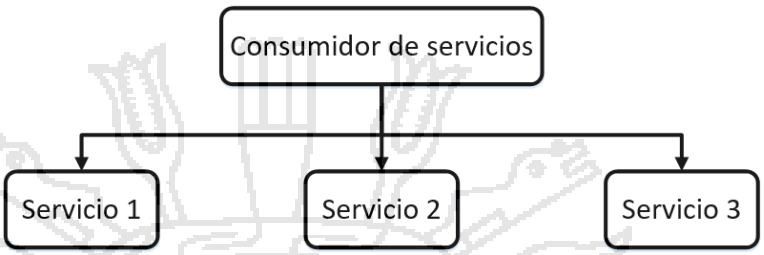


Figura 5. Orquestación de Servicios.
Fuente: (Richards, 2015)

En la Figura 5, se muestra la orquestación de servicios en mención a una analogía de la orquestación en pensar en la orquestación de servicios como una orquesta. Un número de músicos están tocando diferentes instrumentos en diferentes momentos, pero todos están coordinados a través de una persona central. De la misma manera, el componente mediador en la orquestación de servicios actúa como director de orquesta, coordinando todas las llamadas de servicio necesarias para completar la transacción comercial.(Richards, 2015).

1.1.2.2 Coreografía

La coreografía de servicios se refiere a la coordinación de múltiples llamadas de servicio sin un mediador central. El término comunicación entre servicios se utiliza a veces conjuntamente con la coreografía de servicios. Con la coreografía de servicios, un servicio llama a otro servicio, que puede llamar a otro servicio y así sucesivamente, realizando lo que también se conoce como encadenamiento de servicio. Los modelos de coreografía de servicio se utilizan para describir las interacciones de servicio desde un punto de vista global. (Baryannis *et al.*, 2010).

En la Figura 6, se muestra la coreografía de servicios, que describe la colaboración de estos para brindar una solución software.

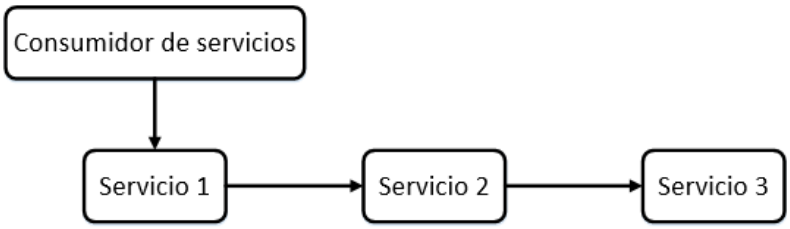


Figura 6. Coreografía de servicios

Fuente: (Richards, 2015).

Newman (2015) menciona que la arquitectura de los microservicios favorece la coreografía de servicios sobre la orquestación de servicios, principalmente porque la topología de la arquitectura carece de un componente de middleware centralizado.

1.1.3 Diferencias entre SOA y microservicios

La arquitectura orientada a servicios (SOA) y la arquitectura de microservicios, están relacionados ya que ambas apuntan a dividir aplicaciones en servicios. No es fácil distinguir entre SOA y microservicios simplemente considerando lo que está sucediendo en la red. Ambos enfoques arquitectónicos tienen servicios que intercambian información a través de la red.

En el nivel de integración aparecen las diferencias entre SOA y la arquitectura de microservicios. En SOA, la solución de integración también es responsable de la orquestación de los servicios, un proceso de negocios se construye a partir de los servicios. En una arquitectura basada en Microservicios, la solución de integración no posee inteligencia. Los Microservicios son responsables de comunicarse con otros servicios.(Eberhard, 2016).

Richards (2015) sostiene que se puede encontrar que el patrón de microservicios es una buena opción de arquitectura inicial en las primeras etapas de su negocio, pero a medida que crece y madura, comienza a necesitar capacidades tales como transformación compleja de solicitudes, orquestación compleja e integración de sistemas heterogéneos. . En estas situaciones, probablemente recurra al patrón SOA para reemplazar su arquitectura inicial de microservicios. Por supuesto, lo opuesto también es cierto: es posible que haya comenzado con una arquitectura SOA grande y compleja, solo para descubrir que no necesitaba todas esas poderosas capacidades

que soporta, en este caso, es probable que se encuentre en la posición común de pasar de una arquitectura SOA a microservicios para simplificar la arquitectura.

1.1.4 Estilo arquitectónico REST

REpresentational State Transfer (REST) es un estilo arquitectónico inspirado en la web con restricciones, este estilo de arquitectura subyacente a la web tiene los siguientes objetivos: como primer objetivo se tiene la escalabilidad de la interacción con los componentes, la web ha crecido exponencialmente sin degradar su rendimiento, una prueba de ello es la variedad de clientes que pueden acceder a través de la web: estaciones de trabajo, sistemas industriales, dispositivos móviles; la generalidad de interfaces, gracias al protocolo HTTP, cualquier cliente puede interactuar con cualquier servidor HTTP sin ninguna configuración especial, esto no es del todo cierto para otras alternativas, como SOAP para los Servicios Web. Como segundo objetivo se tiene la puesta en funcionamiento independiente; este hecho es una realidad que debe tratarse cuando se trabaja en Internet. Los clientes y servidores pueden ser puestas en funcionamiento durante años, por tanto, los servidores antiguos deben ser capaces de entenderse con clientes actuales y viceversa, diseñar un protocolo que permita este tipo de características resulta muy complicado, HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs, a través de la habilidad para crear nuevos métodos y tipos de contenido. Como tercer objetivo se tiene la compatibilidad con componentes intermedios, los más populares intermediaros son varios tipos de proxys para web, algunos de ellos, las caches, se utilizan para mejorar el rendimiento, otros permiten reforzar las políticas de seguridad y otro tipo importante de intermediarios, *gateway*, permiten encapsular sistemas no propiamente Web. Por tanto, la compatibilidad con intermediarios nos permite reducir la latencia de interacción, reforzar la seguridad y encapsular otros sistemas.(Webber, 2010).

Bellido (2015) menciona que REST se diferencia de cualquier otro estilo arquitectónico porque los recursos deben ser identificados de manera única (URI) y los identificadores debe ser opaco para evitar el acoplamiento, es decir, la estructura de un URI no debe incluir ningún significado particular que pueda ser adivinado por un cliente. El estado de los recursos debe ser manipulado a través de operaciones definidas. Un recurso puede admitir varias representaciones que codifican el estado del recurso en un formato determinado.

1.1.5 Tecnologías de contenedores

Cabrera (2016) menciona las diferencias entre una máquina virtual y contenedor de Docker donde indica que una máquina virtual necesita tener virtualizado todo el sistema operativo, mientras que el contenedor Docker aprovecha el sistema operativo sobre el que se ejecuta, compartiendo el Kernel e incluso parte de sus bibliotecas. Para el SO anfitrión, cada contenedor no es más que un proceso que corre sobre el Kernel. El concepto de contenedor o "virtualización ligera" no es nuevo. Otra diferencia es el tamaño, una máquina virtual convencional puede ocupar bastante, sin embargo los contenedores Docker sólo contienen lo que las diferencia del sistema operativo en el que se ejecutan. En cuanto a recursos, el consumo de procesador y memoria RAM es mucho menor al no estar todo el sistema operativo virtualizado.

En la Figura 7, se muestra la comparación entre una máquina virtual y el contenedor de Docker.(Cabrera, 2016).

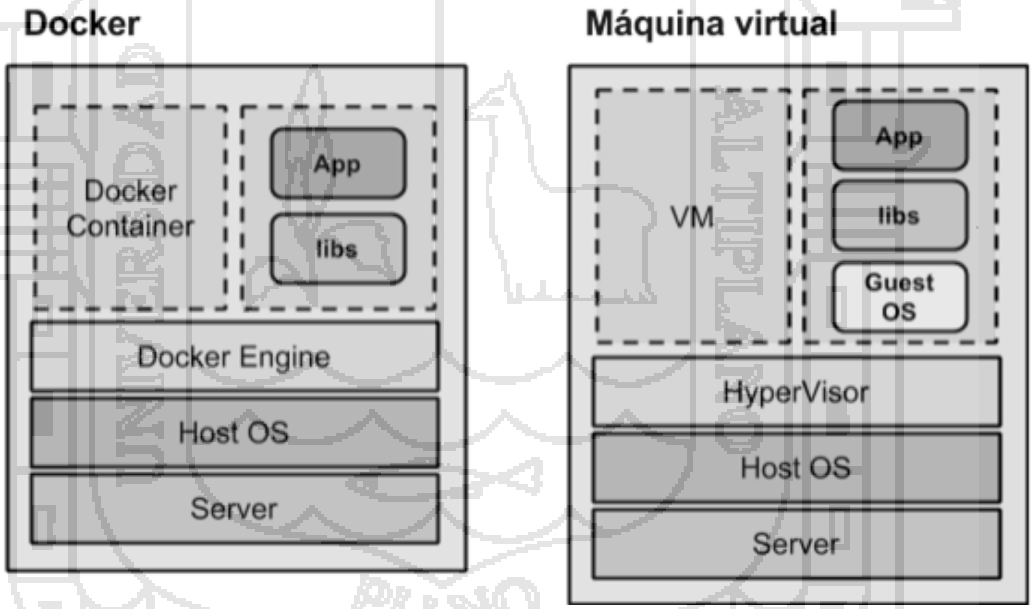


Figura 7. Comparación entre un contenedor Docker y una máquina virtual. Fuente: (Cabrera, 2016).

Los contenedores deben agruparse inteligentemente para formar aplicaciones en funcionamiento y esto requiere una orquestación. La orquestación es donde gran parte de la diferenciación reside en el ecosistema de la contenedorización. Los motores de contenedores como Docker proporcionan herramientas básicas de orquestación. Sin embargo, la orquestación completa implica la programación de

cómo y cuándo deben ejecutarse los contenedores, la administración del clúster y la provisión de recursos adicionales a menudo en múltiples hosts; y es donde herramientas como Kubernetes se adaptan mejor a esta necesidad de orquestar contenedores.(Tarzey, 2015).

1.1.6 Pruebas de carga para aplicaciones web

Las aplicaciones web tienen normalmente muchos usuarios al mismo tiempo, por lo que las solicitudes de muchos usuarios simultáneamente son difíciles de probar; por ello para realizar pruebas efectivas es necesario realizar pruebas automáticas. La prueba de carga es una actividad para probar el rendimiento del sistema en un nivel de carga. El nivel de carga puede ser la cantidad de usuarios a la vez, o la cantidad de datos de procesamiento en línea o la cantidad de peticiones realizadas en un intervalo de tiempo. (Menasce, 2002).

Los criterios de rendimiento en las pruebas de carga propuestos por el autor son:

- *Response Time, es* el tiempo de respuesta, definido como el tiempo empleado por la solicitud de inicio del cliente y la respuesta final del servidor. El tiempo de respuesta es el rendimiento clave del software, para una aplicación web, el tiempo de respuesta de la página se define por el tiempo de red y el tiempo de aplicación.
- *High Availability*, es la alta disponibilidad que brinda una aplicación Web, cuyo objetivo es garantizar un nivel acordado de rendimiento operativo, generalmente tiempo de actividad, durante un período superior al normal. La disponibilidad se refiere a la capacidad de la comunidad de usuarios para acceder y obtener a un servicio, acceder al sistema, ya sea para enviar un nuevo trabajo, actualizar o alterar el trabajo existente, o recopilar los resultados del trabajo anterior.
- El *Throughout*, definido como la cantidad de solicitudes de usuarios procesadas dentro de una unidad de tiempo. Throughout es el rendimiento de carga directa, normalmente se define por hits por segundo, hay dos aspectos del rol; uno se utiliza para diseñar escenarios de pruebas de rendimiento, verificar escenario de pruebas de rendimiento logrado objeto de prueba o no; el otro es analizar el cuello de botella de rendimiento, el límite de todo es el aspecto principal de cuello de botella de rendimiento.

1.2 ANTECEDENTES

Se presenta los antecedentes de investigación en tesis y artículos científicos recientemente publicados en el ámbito global, nacional y local.

Dragoni *et al*. (2017) en su investigación presenta el actual estado de arte de los microservicios, describiendo los problemas abiertos y desafíos futuros respecto a los microservicios. En una de sus conclusiones menciona que debido a que la arquitectura de los microservicios es muy reciente, no se ha encontrado una colección suficientemente amplia de literatura sobre el terreno y se tiene el desafío en el aspecto de seguridad en la comunicación entre los microservicios.

Xiao *et al.* (2017) emprende una investigación crítica de los conceptos clave alrededor de SOA, API y Microservicios, identificando similitudes y diferencias entre ellos; llegando a la conclusión que se debe aprovechar lo mejor de cada concepto para poseer la capacidad de responder a las necesidades del mundo digital.

Aderaldo *et al*. (2017) menciona que aún hay una falta de investigación empírica repetible sobre el diseño, desarrollo y evaluación de aplicaciones de microservicios, y propone el uso de un conjunto inicial de requisitos que pueden ser útiles en la selección de un benchmark de arquitectura.

Pahl y Jamshidi (2016) en su investigación una de sus conclusiones mencionan que los microservicios emergen como un estilo arquitectónico que se extiende desde la etapa del diseño arquitectónico hasta el despliegue.

Kharbuja (2016) en su tesis doctoral tuvo como objetivo proporcionar una comprensión clara de la implementación de la arquitectura de microservicios teniendo como caso de estudio la implementación de un sistema de reserva de habitaciones en línea, en sus conclusiones presenta un conjunto de directrices para la construcción de microservicios.

Garriga *et al*. (2016) en su investigación, examinan un conjunto completo de 29 enfoques de composición de servicios RESTful concluyendo que la composición de servicios RESTful se encuentran en una etapa más temprana de evolución en comparación con la composición basada en SOA. Están pasando de la mera descripción y clasificación de los servicios y las composiciones a características más avanzadas

Ueda *et al*. (2016) en su investigación, analizan el comportamiento de las arquitecturas monolíticas y las arquitecturas basados en microservicios, obteniendo como resultado que el rendimiento del microservicio en un 79,2% es mayor que la monolítica, en la cual se utilizó Docker, como infraestructura para microservicios.

Jaramillo, *et al*. (2016) en su investigación, se tuvo como objetivo implementar un sistema basado en la arquitectura de microservicios a través de un caso de estudio, llegando a la conclusión de que la tecnología Docker puede ayudar eficazmente en el aprovechamiento de la arquitectura microservicios.

Bellido (2015) en su investigación tuvo como objetivo facilitar la composición dinámica de servicios REST a través de un modelo de composición de servicios basado en la calidad del servicio para cual presentó un caso de estudio de un sistema de planificación de viajes. El autor llegó a la conclusión que el modelo propuesto SAW-Q, que sigue los principios del estilo arquitectónico REST, mejora positivamente la calidad de las composiciones dinámicas de servicio.

Savchenko *et al*. (2015) a partir de un estudio de mapeo, tuvo como objetivo proporcionar un análisis de los métodos existentes de pruebas de aplicaciones en la nube basados en microservicios, llegando a la conclusión que los microservicios emergen como un estilo arquitectónico, pero que se extiende desde la arquitectura de la etapa de diseño hacia el despliegue y las operaciones como un estilo de desarrollo continuo. Así mismo concluye que los microservicios están intrínsecamente relacionada con los contenedores basados en la nube para la implementación de los microservicios.

Vivar (2015) realizó una investigación de diferentes arquitecturas de tecnologías orientadas hacia el manejo de servicios; en el que se concluye que las propuestas en el estado del arte respecto a la anotación semántica de servicios, principalmente REST y tecnologías relacionadas provee un enfoque más orientado hacia la automatización.

Steinacker (2015) en su investigación, presenta una introducción a los microservicios, y a través de un caso de estudio muestra cómo las propiedades de pequeñas aplicaciones funcionan bien en las grandes arquitecturas. En una de sus conclusiones menciona que la arquitectura de microservicios admite el trabajo en varios equipos independientes en el que es posible elegir lenguaje de programación, el marco de trabajo o algo similar sin mucho riesgo.

Villamizar *et al*. (2015) en su investigación, analizan y prueban la arquitectura de microservicios y la arquitectura monolítica, en la implementación de una aplicación web en la nube, como resultados se obtuvo que ambas arquitecturas soportan los requisitos comerciales y validan que los microservicios no afectan considerablemente la latencia de las respuestas debido al uso de más hosts.

Richardson (2014) en su investigación presenta motivaciones para el uso de la arquitectura de microservicios y su comparación con las arquitecturas tradicionales, llegando en una de sus conclusiones, que la arquitectura de microservicios tiene ventajas ya que los servicios individuales son más fáciles de entender y pueden desarrollarse e implementarse independientemente de otros servicios.

Brüggemann *et al.*(2014) en su investigación presenta su experiencia con campañas de marketing basadas en correo electrónico donde una coreografía para microservicios ayudó a indicar oportunidades de mejora para la arquitectura y el flujo de información, lo que aumentó el rendimiento.

Liu (2013) en su tesis presenta un estilo arquitectónico de RESTful Service Composition (RSC), que es completamente diferente de los enfoques de orquestación de servicios tradicionales, donde la arquitectura se evalúa en términos de rendimiento, escalabilidad, fiabilidad y modificabilidad; obteniendo como conclusión que los resultados muestran que RSC supera el enfoque de orquestación de servicios.

Karunamurthy *et al*. (2012) en su investigación propone una nueva arquitectura para la composición de servicios web. La arquitectura propuesta amplía el modelo de negocio de servicios web estándar para soportar explícitamente la composición de servicios web.

Pejman *et al*. (2012) en su investigación, presentan varios métodos sobre la composición de servicios centrándose en composiciones de servicios sintácticos basadas en la calidad de servicio.

Fernández (2010) en su investigación presenta un marco de clasificación de servicios ligeros para el estilo arquitectónico REST. En este trabajo de investigación se introdujo los microservicios como una alternativa para simplificar la descripción de servicios, llegando a la conclusión que las descripciones de microservicio son simples y permiten automatizar varias tareas para impulsar la automatización del servicio en la web.

CAPÍTULO II

PLANTEAMIENTO DEL PROBLEMA

2.1 IDENTIFICACIÓN DEL PROBLEMA

En la actualidad, las grandes empresas buscan acercarse más a sus clientes, con este propósito invierten un gran esfuerzo en intentar comprender como se comporta el consumidor, cuáles son sus hábitos, preferencias, gustos u otra necesidad; y esto es reflejado en la gran cantidad de campañas publicitarias y diversas estrategias de ventas, que son elementos que se han utilizado durante mucho tiempo; sin embargo con el crecimiento tecnológico acelerado en el que se ha estado viviendo, las empresas tienen que adaptarse a los cambios tecnológicos y explotar más a fondo la forma de utilizar estas tecnologías.

El comercio electrónico en el Perú ha crecido, en un 198% en los últimos años y con ello las aplicaciones web, permitiendo realizar comercio electrónico que consiste en la compra y venta de productos o servicios a través de la Internet; éstas aplicaciones facilitan a los clientes la manera de realizar sus compras y a las empresas incrementar sus ventas; ocasionando una demanda de usuarios concurrentes en un tiempo determinado.(CAPECE, 2016). Esta demanda trae consigo problemas en el rendimiento, la disponibilidad continua y permanente; y el tiempo de respuesta cuando los usuarios intentan acceder a las funcionalidades de la aplicación web.

Con el afán de explotar estas oportunidades de comercialización, las grandes empresas realizan inversiones en la implementación de aplicaciones web, con el fin de ser más competitivas, para mantenerse activo y competitivo en el negocio, pero acercarse no es suficiente, la clave es ofrecer un valor agregado a sus clientes brindando un servicio de calidad y es donde la ingeniería de software busca una forma de satisfacer los diferentes requerimientos del negocio que son cada vez más exigentes.

Las diferentes empresas van adoptando el cambio de paradigma en el desarrollo de software, con la finalidad de enfatizar más en el proceso de negocio a través de arquitecturas agiles y flexibles, adaptables a los cambios continuos que ocurre en los modelos de negocios de estas organizaciones (Cockcroft *et al*., 2016).

En el Perú, de la totalidad de las empresas, el 13.6% desarrolla su propio software ya que muestran un modelo de negocio dinámico que tiene que ser reflejado en el software que utilizan (INEI, 2016), donde actualmente la construcción de software con arquitecturas monolíticas es común entre las diferentes empresas, pero cuando existe los desafíos de la escalabilidad, la flexibilidad, el desarrollo rápido, el corto tiempo de lanzamiento al mercado y la colaboración del equipo de desarrollo, que son cada vez más amplios y variados; la construcción de software, en específico la elección de la arquitectura es un punto crítico y complejo para lograr la competitividad empresarial.

Para llevar a cabo la tarea de diseñar y desarrollar software que pueda soportar y escalar este tipo de situaciones de crecimiento continuo y circunstancias cambiantes, se han venido utilizando diferentes estilos arquitectónicos de software, entre ellos, uno de los más usados ha sido el estilo arquitectónico por capas, conocido como el modelo monolítico, el cual ha funcionado muy bien para la mayoría de problemas de pequeñas y medianas empresas, pero con limitaciones para las situaciones que existen en la actualidad. (Mazlami, *et al*., 2017).

La Arquitectura Orientada al Servicio (SOA) es un modelo en el que la lógica de una aplicación se descompone en varias unidades más pequeñas que conjuntamente implementan la lógica del negocio. SOA promueve que estas unidades individuales existen independientemente pero no aisladas entre sí, a su vez podría ser distribuidas; sin embargo se cuenta con problemas relacionado a protocolos de comunicación, que es la lógica de intercambio de comunicación entre aplicaciones de proveedor, falta de orientación sobre la granularidad de los servicios u orientación sobre la selección de servicios para dividir el sistema. La arquitectura SOA se suele describir como aquellos componentes de aplicación que se comunican para proporcionar servicios a otros componentes a través de una red. El objetivo de SOA era crear aplicaciones distribuidas resistentes sin complejos componentes centralizados. Sin embargo, SOA se acopla estrechamente a las estructuras organizativas y se aplicaba como soporte

de las nuevas estructuras internas. Por lo tanto, su éxito depende en gran medida de las capacidades organizativas reestructuradas, resultantes y en la estructura de los equipos que diseñaban la arquitectura. En lugar de crear sistemas de bajo acoplamiento pero autónomos, SOA crea sistemas muy frágiles que precisaban de una infraestructura compleja. (Newman, 2015).

La arquitectura del microservicio es un estilo que ha ido ganando popularidad cada vez más en los últimos años, tanto en la literatura de la arquitectura del software , que se evidencia en los congresos de ingeniería de software, y en el ámbito empresarial, donde las empresas reconocidas tales como Netflix, Twitter entre otras han adoptado la arquitectura de microservicios. (Newman, 2015).

Actualmente se cuenta con técnicas de composición de servicios (Bellido, 2015), pero enfocadas a los servicios tradicionales, así mismo existe una limitada literatura de composición de microservicios (Dragoni *et al*., 2017). Incluso se cuenta con pocas experiencias en el desarrollo de sistemas basados en microservicios por lo que no se tiene una referencia clara del proceso del desarrollo de aplicaciones basadas en la arquitectura de microservicios.(Kharbuja, 2016). Por otro lado se cuenta con la tecnología de contenedores basados en la nube y que éstos se encuentran estrechamente relacionados con la arquitectura de microservicios (Savchenko *et al*., 2015).

2.2 ENUNCIADOS DEL PROBLEMA

Al identificar el problema, desde la perspectiva de la arquitectura de software existen modelos arquitectónicos para satisfacer esta necesidad contextual y tecnologías emergentes en cuanto a infraestructura tecnológica. En la presente investigación se considera la arquitectura de microservicios, la cual cuenta con casos de éxito, de los cuales se puede tomar de referencia para enfocarlo en el ámbito nacional y regional, en este caso al modelo del negocio de comercio electrónico.

Por tanto en la presente investigación se realizó la siguiente formulación del problema: ¿Es posible implementar un modelo de composición de microservicios para la implementación de una aplicación Web de comercio electrónico utilizando la herramienta tecnológica Kubernetes?

2.3 JUSTIFICACIÓN

La arquitectura de software proporciona un marco de referencia para organizar un sistema en subsistemas, componentes o módulos. Con la arquitectura del software también se define la interacción entre sus componentes individuales. Por tanto definir una arquitectura es una parte importante en el desarrollo de software que le permite tener una vista global del sistema software a implementar.

Una de las desventajas de las aplicaciones con arquitectura monolítica, arquitectura basado en capas, ocurre cuando se necesitan subir cambios a una aplicación Web que se encuentra alojada en un servidor, para esto es necesario detener todos los procesos que se están realizando dentro de esta aplicación, así el cambio o los cambios no tengan nada que ver con los procesos realizados en ese momento. Otra de las desventajas de las aplicaciones con este tipo de arquitectura es la duplicación del código ya que al ser aplicaciones monolíticas no comparten entre ella las funcionalidades de sus módulos, para esto se visionó hace varios años una arquitectura que permitiera estar hecha de pequeños servicios que se pudieran juntar para cumplir la necesidad de uno o muchos clientes.

Conforme ha pasado el tiempo la tecnología ha ido creciendo de forma exponencial y hoy en día se cuenta con recursos e infraestructuras tecnológicas como servidores con grandes capacidades que se pueden adaptar con clústeres, diferentes protocolos de comunicación como REST, que hacen posible una comunicación fácil de implementar con arquitectura basadas en microservicios. Por otro lado dada la gran demanda por parte de empresas multinacionales de la talla de Netflix, Google, Amazon, entre otras, y el inicio de la implementación de este estilo arquitectónico en empresas nacionales, hace constatar que existe un mercado emergente por parte de los equipos de desarrollo de software.

A nivel teórico, existe limitada literatura de composición de microservicios, lo que motivó a la presente investigación para proponer un modelo de composición de microservicios, de esta manera se pretende aportar a la literatura sobre la composición de microservicios a nivel de infraestructura tecnológica utilizada y a nivel de servicios basados en el diagrama de procesos del modelo de negocio.

En la presente investigación que propone un modelo de composición de microservicios tendrá como caso de estudio el desarrollo de una aplicación Web de

comercio electrónico basado en la arquitectura de microservicios y utilizando Kubernetes, como herramienta tecnológica de gestión de contenedores, donde el proceso de desarrollo de la aplicación conformado por el análisis, diseño, implementación y evaluación de la composición de microservicios servirá como referencia a los desarrolladores de software de sistemas distribuidos de las empresas regionales y nacionales.

En la presente investigación se utilizó la tecnología de contenedores, como infraestructura tecnológica en la nube, en específico la herramienta tecnológica Docker para el despliegue de los microservicios y Kubernetes como manejador de aplicaciones en contenedores; que permite a los equipos de desarrollo de software, investigadores u otros interesados, conocer su arquitectura, configuración y funcionamiento en la implementación de la aplicación Web de comercio electrónico bajo el modelo propuesto.

Con la presente investigación, también se pretende que las oficinas o áreas de tecnología e informática de las empresas regionales y nacionales puedan considerar como referencia el modelo propuesto, basado en la arquitectura de microservicios, para el desarrollo e innovación de sistemas distribuidos en la nube, migrando las aplicaciones que fueron implementadas en forma monolítica a la arquitectura de microservicios, donde los equipos de desarrollo puedan realizar un mejor seguimiento de la integración continua, los despliegues en producción utilizando tecnología independiente como la elección de un lenguaje de programación o un marco de trabajo.

Con los resultados de la presente investigación se puede realizar futuras investigaciones aplicando el modelo propuesto de composición de microservicios para el desarrollo de aplicaciones web o móviles de comercio electrónico u otros modelos de negocio.

Finalmente con la presente investigación se pretende motivar a realizar futuras investigaciones empleando otras herramientas tecnológicas que faciliten la composición de microservicios para la implementación de una aplicación web de otros modelos de negocio. Así mismo realizar comparativas con la arquitectura monolítica en los indicadores de tiempo de respuesta, disponibilidad y rendimiento.

2.4 OBJETIVOS

2.4.1 Objetivo general

Proponer un modelo de composición de microservicios para la implementación de una aplicación web de comercio electrónico utilizando Kubernetes.

2.4.2 Objetivos específicos

- Analizar los requerimientos arquitectónicos de la aplicación web de comercio electrónico.
- Diseñar un modelo de composición de microservicios para la implementación de una aplicación web de comercio electrónico.
- Implementar el modelo de composición de microservicios para una aplicación web de comercio electrónico.
- Evaluar el modelo de composición de microservicios para la implementación de una aplicación web de comercio electrónico.

2.5 HIPÓTESIS

2.5.1 Hipótesis General

El modelo de composición de microservicios para la implementación de una aplicación Web de comercio electrónico utilizando Kubernetes funciona significativamente.

2.5.2 Hipótesis Específicas

- El tiempo de respuesta de la aplicación web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es menor o igual que el modelo monolítico.
- La disponibilidad de la aplicación web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es mayor que el modelo monolítico.
- El rendimiento de la aplicación web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes, es mayor que el modelo monolítico.

CAPÍTULO III

MATERIALES Y MÉTODOS

3.1 LUGAR DE ESTUDIO

La presente investigación se realizó en la región de Puno – Perú. La implementación de la aplicación web de comercio electrónico basado en el modelo de composición de microservicios se desplegó y probó en un ambiente virtual, en específico en una plataforma en la nube, que es Google Cloud Plataform, dónde se proporciona una buena infraestructura tecnológica a nivel de hardware.

3.2 POBLACIÓN

La población estuvo conformada por las peticiones de consumo a la aplicación web de comercio electrónico, que fueron expresadas en cargas de trabajo en un determinado tiempo.

3.3 MUESTRA

La selección de la muestra fue de tipo no probabilístico donde se utilizó el muestreo por conveniencia, este tipo de muestreo se caracteriza por obtener muestras accesibles representativas. Por tanto se consideró como muestra 20 peticiones por segundo, estas peticiones de consumo fueron simuladas por la herramienta tecnológica Locust.

3.4 MÉTODO DE INVESTIGACIÓN

La presente investigación es del tipo experimental tecnológico porque manipula directamente la variable independiente, el modelo de composición de microservicios, para para medir los efectos en la variable dependiente, la implementación de una aplicación web de comercio electrónico, este método se aplica con el propósito de establecer las conclusiones y generalizar los resultados de la investigación en forma cuantitativa.

3.5 MÉTODOS EMPLEADOS

3.5.1 Metodología XP

Una metodología ágil es una metodología de gestión de proyectos que utiliza ciclos de desarrollo cortos para centrarse en la mejora continua del desarrollo de un producto o servicio, más que centrarse en la gestión del propio proyecto. La metodología XP (Xtreme Programing) o metodología de programación extrema, es una metodología ágil para el desarrollo de software, que se centra en la programación de la solución software.

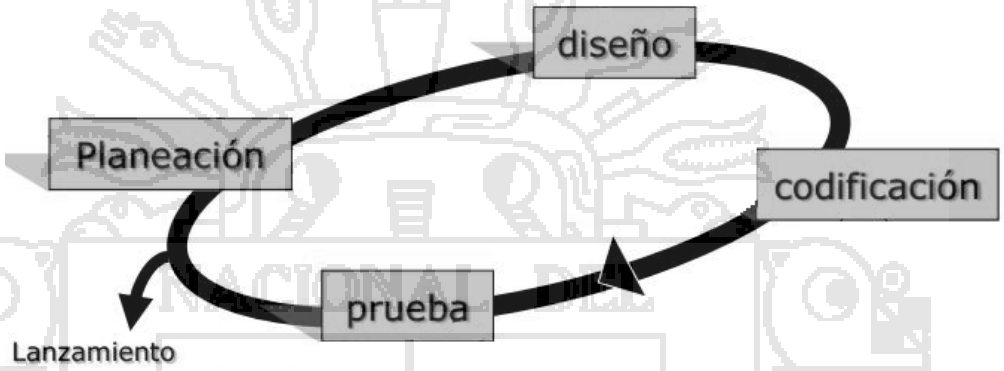


Figura 8. Fases de la metodología XP.
Fuente: Sánchez (2007).

Sánchez (2007) menciona que el ciclo de vida de un proyecto XP es flexible y dinámico y considera las siguientes fases que se muestra en la Figura 8 y se detallan a continuación.

La fase de planeación que define las historias de usuario, que son descripciones sobre las características que debe tener el sistema software sin hacer mucho hincapié en los detalles; estas historias de usuario constan de tres o cuatro líneas, escritas en un lenguaje no técnico, de modo que sea comprensible y memorizable por el equipo de desarrollo del software.

La fase de diseño la metodología XP sugiere que hay que realizar diseños simples y sencillos procurando hacerlo lo menos complicado posible para conseguir un diseño fácilmente entendible e implementable, que durante el proceso costará menos tiempo y esfuerzo desarrollar.

La fase de codificación se concentra en tener estándares de codificación para mantener el código consistente y fácil de leer y volver a factorizar, esto ayuda a los desarrolladores a comprender los requisitos. La programación de pares es una de las prácticas que distinguen la metodología XP. Cada par de programadores escribe su código y luego lo integran.

La fase de pruebas, que es uno de los pilares de la metodología XP, es el uso de pruebas para comprobar el funcionamiento del Software que se implementó.

3.5.2 Historia de usuario y Tarea de ingeniería

En la presente investigación para el análisis de los requerimientos, de la implementación de la aplicación Web de comercio electrónico, se realizó con las técnicas de historias de usuario y tareas de ingeniería.

La historia de usuario, es una técnica para la especificación de requerimientos Software, a través de una descripción breve y comprensible del comportamiento que se desea del Software, estas historias de usuario se realizan por cada característica principal del Software y reemplazan un documento complejo de requisitos. El formato para la elaboración de las historias de usuario se muestra en la Figura 9, con la explicación de cada uno de sus componentes.

HISTORIA DE USUARIO	
Número: Permite identificar a una historia de usuario.	**Usuario:** Persona que utilizará la funcionalidad del sistema descrita en la historia de usuario.
Nombre de historia: Describe de manera general a una historia de usuario.	
Como [Rol] quiero: Describe de manera general a una historia de usuario.	
Para: Describe el motivo de su requerimiento	

Figura 9. Formato de historias de usuario.

Fuente: (Letelier y Penadés, 2006).

Una historia de usuario se descompone en una o varias tareas de ingeniería, las cuales describen las actividades que se realizarán en cada historia de usuario, considerando las estimaciones en tiempo, así mismo las tareas de ingeniería se vinculan más al desarrollador del software, ya que permite tener un acercamiento con el código. El formato a utilizarse para la elaboración de las tareas de

ingeniería se muestra en la Figura 10, con la explicación de cada uno de sus componentes en un lenguaje más técnico.

TAREA DE INGENIERÍA	
Número de Tarea: Permite identificar a una tarea de ingeniería.	**Número de Historia:** Número asignado de la historia correspondiente
Nombre de Tarea: Describe de manera general a una tarea de ingeniería.	
Tipo de Tarea: Tipo al que corresponde la tarea de ingeniería.	**Puntos estimados:** Número de días que se necesitará para el desarrollo de una tarea de ingeniería.
Fecha de Inicio: Fecha inicial de la creación de la tarea de ingeniería.	**Fecha Fin:** Final concluida de la tarea de ingeniería.
Programador Responsable: Persona encargada de programar la tarea de ingeniería.	
Descripción: Información detallada de la tarea de ingeniería.	

Figura 10. Formato de tarea de ingeniería.

Fuente: (Letelier y Penadés, 2006).

3.5.3 Diagramas arquitectónicos

En la presente investigación se utilizó diagramas arquitectónicos, que son diagramas que expresan cuáles son los componentes; físicos, lógicos y la relación de éstos; que conforman el sistema Software. El diagrama de la arquitectura física expresa los componentes físicos, como por ejemplo una terminal, dispositivo móvil, servidor y similares, que participan en el modelo de composición de microservicios propuesto, la relación entre ellos y la explicación de los mismos. Así mismo el diagrama de la arquitectura lógica expresa cuáles son los componentes lógicos, que se refieren a los subsistemas, módulos de la solución software; y la relación entre ellos, que participan en el sistema software. La especificación de esta arquitectura es similar a la arquitectura física, se especifican los componentes y las relaciones entre ellos.

3.5.4 Algebra de composición de servicios

En la presente investigación se utilizó el álgebra de composición de servicios para componer las funcionalidades de cada microservicio, y tiene la siguiente notación: (Hamadi y Benatallah, 2003).

$$S \coloneqq (S \diamond S) \mid S \odot S \mid S /\!/ S|$$

Donde:

S : *Servicio.*

$/\!/$: *Invocación paralela.*

$\diamond$: *Invocación alternativa.*

$\odot$: *Invocación secuencial.*

3.5.5 Programación orientada a objetos

En la presente investigación, para la implementación del modelo de composición de microservicios propuesto, se empleó la programación orientada a objetos, que es un paradigma de programación donde se abstrae entidades como objetos y éstos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad específica.

3.5.6 Métricas de calidad de Microservicios

En la presente investigación, para la evaluación de los microservicios implementados, se utilizó las métricas de calidad empleados, considerándose como atributos de calidad: la granularidad, el acoplamiento, la cohesión, la complejidad y la reutilización (Kharbuja, 2016).

La granularidad de un servicio se describe como el tamaño adecuado de un servicio, es evaluada como la razón del cuadrado de número de operaciones de un servicio y el cuadrado del número de mensajes consumidos por el servicio y está dado por:

$$granularidad = \frac{número\ de\ operaciones}{número\ de\ mensajes} = \frac{[O(s)]^2}{[M(s)]^2}$$

Donde:

$O(s)$: Es el conjunto de operaciones proporcionado por un servicio.

$M(s)$: Es el conjunto de mensajes de todas las operaciones de un servicio.

El acoplamiento de un servicio se describe como la fuerza de la dependencia entre servicios en el sistema, es evaluado como el promedio del número de servicios conectados directamente y está dado por:

$$acoplamiento = \frac{número\ de\ servicios\ conectado}{número\ de\ servicios} = \frac{n_c + n_p}{n_s}$$

Donde:

n_c: Número de servicios consumidores.

n_p: Número de servicios dependientes.

n_s: Número total de servicios.

La cohesión se describe la fuerza de la relación entre las operaciones en un servicio, es evaluado como el inverso del promedio del número de mensajes usados, y está dada por:

$$cohesión = \frac{n_s}{M(s)}$$

Donde:

$M(s)$: Es el conjunto de mensajes de las operaciones de un servicio.

n_s: Número total de servicios.

La complejidad se describe como la dificultad de entender la relación entre los servicios, es evaluado como el número de operaciones y está dada por:

$$complejidad = \frac{granularidad\ del\ servicio}{número\ de\ servicios}$$

$$= \frac{\sum_{i=1}^{O(s)} (SG(i))^2}{S}$$

Donde:

S: Número de servicios.

$SG(i)$: Granularidad de i-ésimo servicio.

$O(s)$: Número de operaciones del servicio.

La reusabilidad se describe como la capacidad de reutilización de un determinado microservicio se describe como es evaluada como el número de consumidores existentes del servicio y está dada por:

$$resusabilidad = número\ de\ consumidores\ existentes$$

$$= S_{consumidores}$$

Donde:

$S_{consumers}$: Consumidores del servicio.

Kharbuja (2106) propone las métricas básicas de calidad, que fueron consideradas para la evaluación de los atributos de calidad de los microservicios implementados en la presente investigación, éstas métricas son las siguientes: número de operaciones proporcionadas por un servicio, número de mensajes de las operaciones de un servicio, número de servicios, número de servicios conectados, número de servicios consumidores, número de servicios dependientes y el número de consumidores del servicio.

3.5.7 Pruebas de Carga

Para la evaluación del modelo de composición de microservicios, para la implementación de una aplicación Web de comercio electrónico, se empleó la técnica de pruebas de carga que se refiere a la práctica de evaluar el comportamiento del sistema bajo carga. Una carga es la tasa de peticiones por segundo entrantes al sistema. En la presente investigación se utilizó las tres fases de una prueba de carga propuesto por Jiang (2015) que consiste en el diseño, la ejecución y el análisis de una prueba de carga.

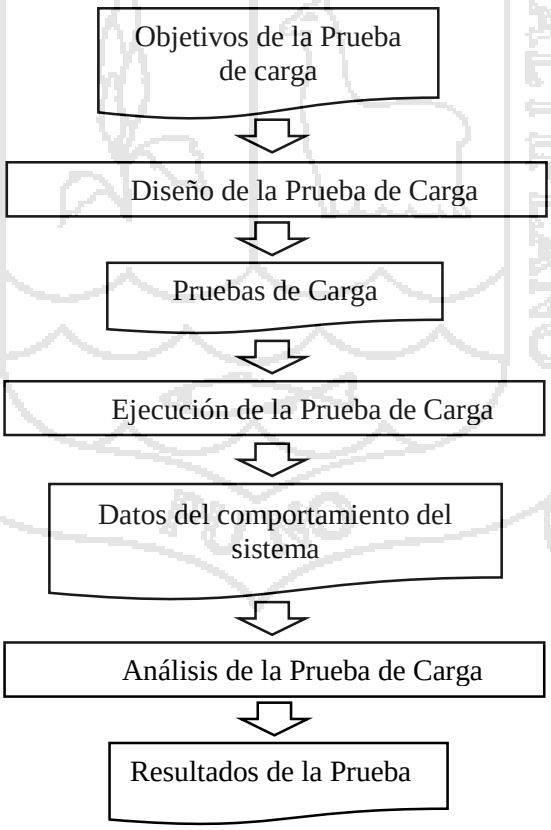


Figura 11. Descripción general del proceso de Pruebas de Carga
Fuente: (Jiang, 2015).

En la Figura 11, se muestra las fases del proceso de pruebas de carga, además se muestra el resultado o salida de cada fase que son las pruebas de carga, los datos del comportamiento del sistema y los resultados de la prueba.

3.6 MATERIALES EMPLEADOS

Para desarrollar el presente trabajo de investigación se utilizó las siguientes herramientas tecnológicas; que son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software, que fueron utilizadas para el cumplimiento de los objetivos.

Estas herramientas fueron seleccionadas por ser software libre y las que se adaptaron mejor en el desarrollo del modelo de composición de microservicios propuesto.

3.6.1 Kubernetes

Para la implementación del modelo de composición de microservicios se utilizó como infraestructura tecnológica a Kubernetes, que es un sistema de orquestación de contenedores, un contenedor es como una máquina virtual ligera. En la Figura 12 se ilustra la arquitectura de Kubernetes que sigue un enfoque maestro-esclavo.

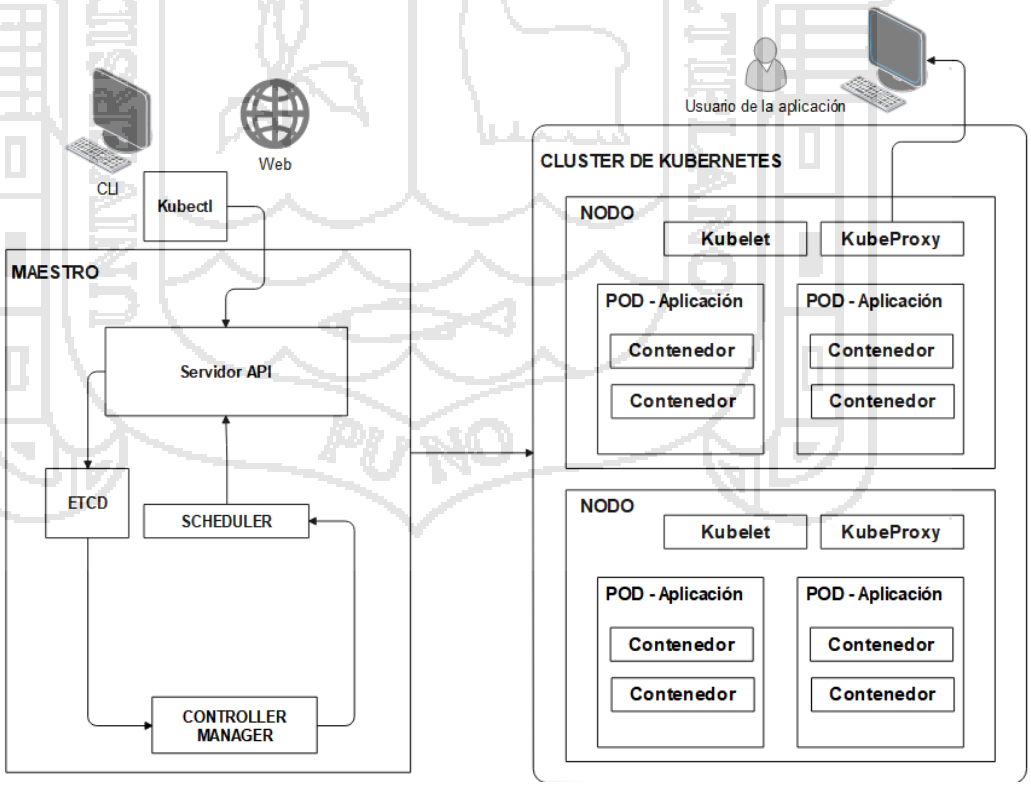


Figura 12. Arquitectura de Kubernetes.

Fuente: (Oliveira, Lung, Netto, y Rech, 2017)

El maestro de Kubernetes es el plano de control de la arquitectura, quién es el responsable de la planificación de despliegues, actuando como puerta de enlace para la API y para la administración general del clúster que es un conjunto de máquinas virtuales o servidores físicos.

El maestro consta de un servidor API, un *scheduler, ETCD* y un *controller manager*. El maestro es responsable de la programación global a nivel de grupo de pods y del manejo de eventos. El servidor API, es el servidor principal que se ejecuta en el maestro, alberga un servicio REST que se puede consultar para mantener el estado deseado del clúster y para mantener las cargas de trabajo. El *Scheduler*, es el servicio planificador que se utiliza para programar cargas de trabajo en contenedores que se ejecutan en los nodos trabajadores, funciona junto con el servidor API para distribuir aplicaciones entre grupos de contenedores que trabajan en el clúster. El *controller manager*, es un administrador que controla una variedad de funciones de clúster, monitorea, administra y descubre nuevos nodos, también administra y actualiza los *endpoints*, que son los puntos finales del servicio. El ETCD es el almacén de clave valor distribuido que proporciona una forma confiable de almacenar datos en un conjunto de máquinas.

Un concepto importante en la arquitectura de Kubernetes es el nodo, que se refiere a máquinas virtuales o servidores físicos. El nodo es el esclavo en la arquitectura y ejecuta los componentes de la pila de aplicaciones, que se denomina Pod. Cada nodo ejecuta varios componentes de Kubernetes, como un Kubelet y un KubeProxy. El Kubelet es un proceso agente que funciona para iniciar y detener grupos de contenedores que ejecutan aplicaciones de usuario. El KubeProxy funciona como un servicio de red proxy que redirige el tráfico a servicios y grupos específicos. Ambos agentes se comunican con el maestro a través del servidor API.

El Pod, unidad de infraestructura que ejecuta una aplicación, es un grupo de uno o más contenedores que se ejecutan en el mismo Host, se programan juntos y comparten una configuración común de dirección IP / puerto.

Los nodos se unen para formar clúster que el maestro utiliza para ejecutar componentes de la aplicación. Los nodos alojan aplicaciones de usuario final

utilizando sus recursos locales, como computación, red y almacenamiento. Por lo tanto, incluyen componentes para ayudar en el registro, descubrimiento de servicios, etc.

La mayoría de las interacciones administrativas y de control se realizan mediante el *script* Kubectl, que es un archivo donde contiene instrucciones para ser ejecutadas, o realizando llamadas REST al servidor API. El estado del clúster y las cargas de trabajo que se ejecutan en él se sincronizan constantemente con el maestro utilizando todos estos componentes.

3.6.2 Docker

Docker es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones en contenedores de software, proporcionando una capa de abstracción y automatización de virtualización a nivel de sistema operativo en Linux.

En la presente investigación se utilizó herramienta Docker para empaquetar los microservicios con todas sus dependencias en una unidad estandarizada para el desarrollo de software. Así mismo Docker permitió separar la aplicación de la infraestructura permitiendo enviar, probar y desplegar códigos rápidamente a un contenedor.

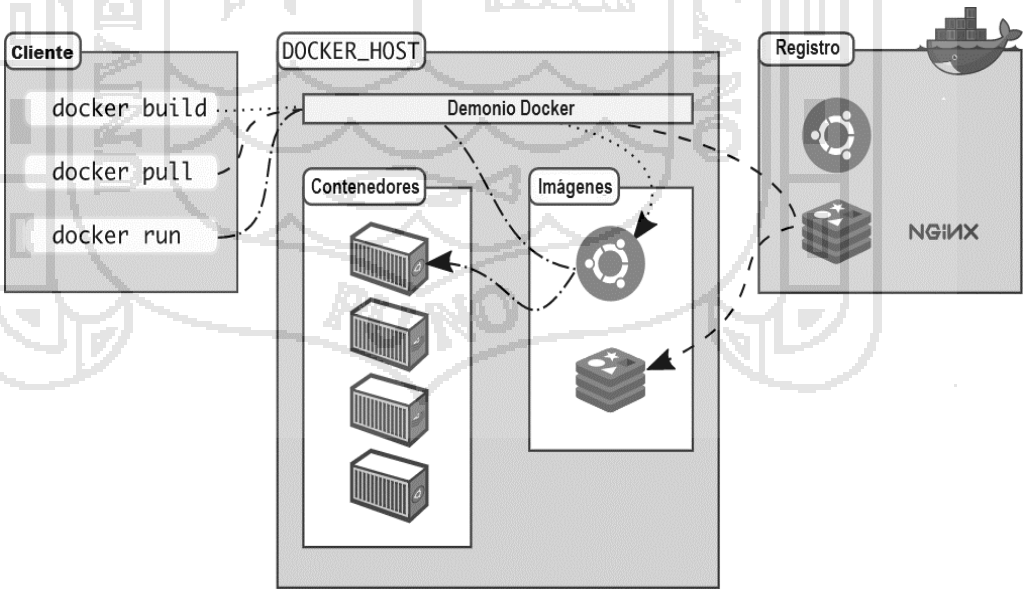


Figura 13. Arquitectura Docker.

Fuente:(Docker Inc., 2017).

En la Figura 13, se muestra la arquitectura Docker, donde el cliente de Docker es la principal interfaz de usuario para Docker, a través de él se envió comandos para que se comunique con el demonio Docker. El demonio Docker corre en una máquina anfitriona llamada host. El demonio Docker levanta los contenedores haciendo uso de las imágenes que fueron almacenadas en el registro de Docker. Cada contenedor se creó a partir de una imagen y es un entorno aislado y seguro dónde se ejecuta el microservicio.

3.6.3 Locust

En la presente de investigación para la elaboración de las pruebas de carga se utilizó Locust, que es herramienta software moderna para realizar prueba de carga para aplicaciones Web, es de código abierto y escrita en lenguaje de programación Python.

3.6.4 Bizagi Process Modeler

Bizagi Process Modeler es una herramienta *software* utilizado para diagramar, documentar y simular procesos usando la notación estándar BPMN (Business Process Modeling Notation). En la presente investigación se utilizó ésta herramienta para el modelado del proceso de negocio de comercio electrónico.

3.6.5 Echo

Echo es un marco de trabajo para aplicaciones Web creado por la empresa NextApp. Con Echo las aplicaciones se desarrollan utilizando una API orientada a los componentes y orientada a eventos, lo que elimina la necesidad de lidiar con la naturaleza basada en páginas de los navegadores. En la presente investigación se utilizó Hecho por el conjunto de herramientas y librerías que cuenta para la implementación del lado del servidor.

3.6.6 Vue.js

Vue.js es un marco de trabajo de código abierto progresivo de JavaScript, que es un lenguaje de programación, para construir interfaces de usuario. Con Vue.js la integración en proyectos que usan otras bibliotecas de JavaScript es más fácil. En la presente investigación se utilizó Vue.js porque está diseñada para ser adoptable incrementalmente para implementación de interfaces para el lado del cliente.

3.6.7 Golang

Go es un lenguaje de programación concurrente y compilado inspirado en la sintaxis del lenguaje de programación C. Ha sido desarrollado por Google. En la presente investigación la codificación de la aplicación Web de comercio electrónico del lado del servidor fue realizado con este lenguaje por ser tendencia en los lenguajes de programación más utilizados en la programación de sistemas *software*.

3.6.8 JWT

JSON Web Token (JWT) es un estándar abierto que define una forma compacta y autónoma para transmitir información de forma segura entre las partes como un objeto JSON. Esta información puede ser verificada y confiable porque está firmada digitalmente. Los JWT se pueden firmar usando un claves públicas/ privadas usando RSA (algoritmo de clave pública).

En la presente investigación se utilizó JWT en la autenticación y para el intercambio de información entre los microservicios. En la autenticación, una vez que el usuario haya iniciado sesión, cada solicitud posterior incluyó el JWT, lo que permitió acceder a las rutas, servicios y recursos permitidos. En el intercambio de información se empleó para transmitir información de forma segura entre los microservicios.

3.6.9 NGINX

NGINX es un software de código abierto para servicios Web, proxy inverso, almacenamiento en caché, balanceo de carga, transmisión de medios y más. El software NGINX está fuertemente asociado con los microservicios, ya sea que se utilice para implementar como un proxy inverso, o como un servidor Web altamente eficiente facilitando el desarrollo de aplicaciones basado en microservicios y mantiene las soluciones funcionando sin problemas. En la presente investigación se utilizó NGINX como un servidor proxy inverso por ser altamente eficiente.

CAPÍTULO IV
RESULTADOS Y DISCUSIÓN

En este capítulo se presenta los resultados y discusión de acuerdo a los objetivos específicos.

4.1 RESULTADOS CONFORME AL OBJETIVO ESPECÍFICO 1

Los requisitos de la arquitectura contienen una visión general del software y se expresó a través requerimientos funcionales y de requerimientos de atributos de calidad.

4.1.1 Historias de usuario y tareas de ingeniería

Para el análisis de éstos requerimientos se realizó el modelado del proceso de negocio de compras online que se detalla en el Anexo 1, seguidamente se realizó las historias de usuario, que se detalla en el Anexo 2, las tareas de ingeniería se detalla Anexo 3.

Las tareas de ingeniería se identificaron a partir de las historias de usuario y a diferencias de éstas contiene descripciones de las características del sistema en un lenguaje técnico, y éstas son:

Se identificó que el cliente requiere que la aplicación web, pueda ser accedida en cualquier momento debido a que su mercado objetivo no solo está dado por el del mismo país sino a nivel global, por tanto se tiene usuarios en otros países con diferente zona horaria; estos usuarios requieren la alta disponibilidad de la aplicación Web de comercio electrónico.

El cliente muestra preocupación por la seguridad del sistema y es expresada en la historia de usuarios, esta preocupación es a nivel de transacciones, puesto que el usuario podrá realizar compras en línea y en estas operaciones se hace manejo del dinero, por lo cual se toma en cuenta los tokens para las transacciones y adicionalmente se realiza la encriptación de los datos que viajan por el medio de internet.

El cliente también ha expresado a través de la historia de usuario, que el software debe ser a medida, así mismo requiere que su software sea escalable debido al dinamismo de su modelo de negocio que viene realizando y reducir el tiempo de respuesta al realizar mantenimiento de software, de forma que el cambio de modelo de negocio no se vea delimitado por el software, es en esta parte que se buscó implementar una arquitectura altamente escalable y así mismo pueda incorporar nuevas funcionalidades.

El cliente expresa el requerimiento, en el acceso concurrente de usuarios a la aplicación web, técnicamente esto depende de la arquitectura en la que el servicio pueda atender las peticiones concurrentes y esta arquitectura debe estar sobre una infraestructura hardware que pueda soportar gran cantidad de peticiones concurrentes.

Finalmente se identificó que el cliente requiere el rendimiento del software, puesto que el sistema tiene que atender peticiones concurrentes de las funcionalidades ofrecidas, por ello se buscó una arquitectura que pueda soportar estos requerimientos que fue tomado en cuenta en el momento de la implementación, específicamente al momento de diseñar la arquitectura.

4.1.2 Casos de uso

Mediante los casos de uso se identificó los requerimientos funcionales y se describió las actividades a realizarse para llevar a cabo el proceso de comercio electrónico.

En la Figura 14 se muestra, el resumen de las relaciones entre los casos de uso, los actores y el sistema.

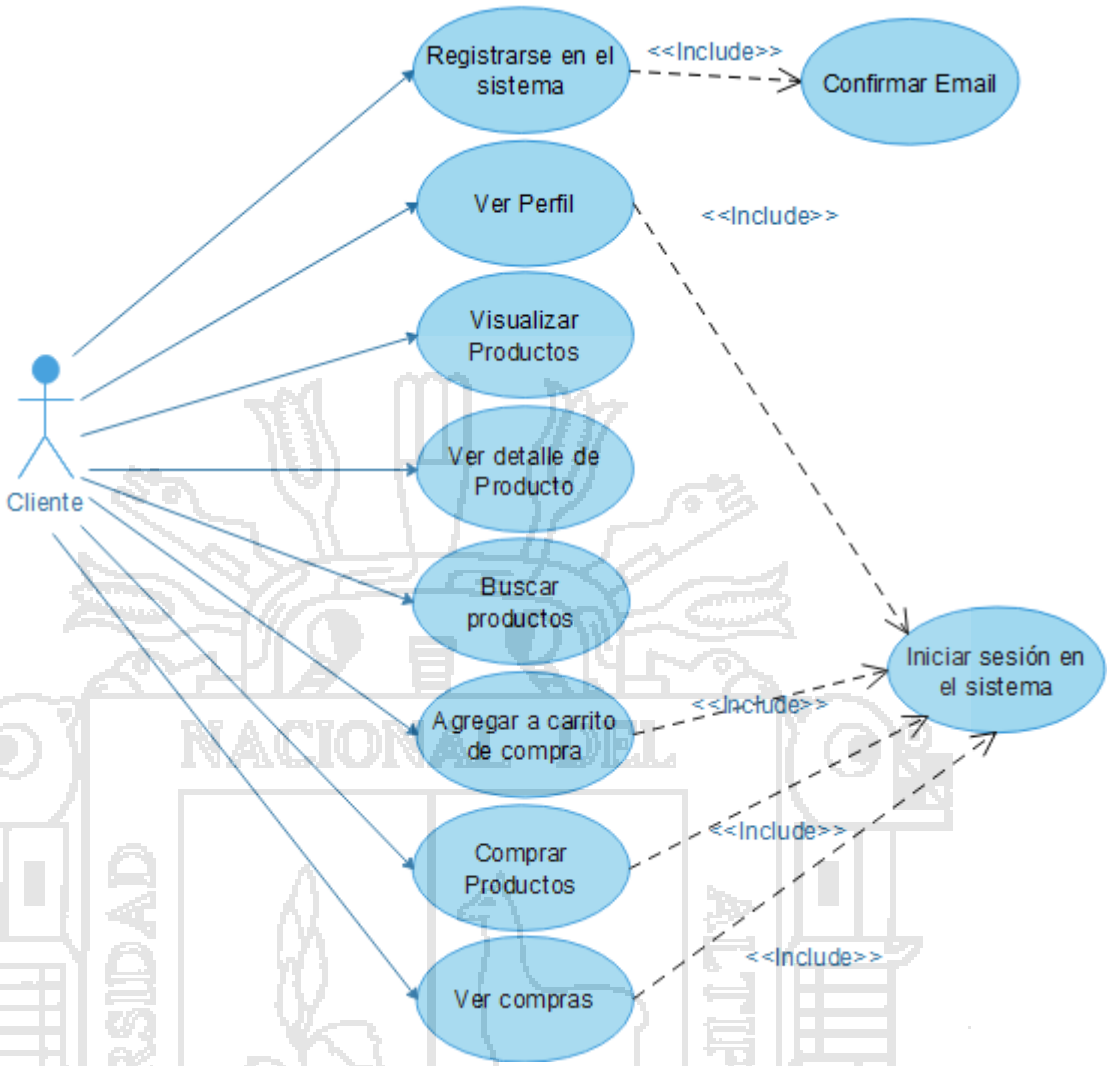


Figura 14. Caso de uso del proceso de compras en línea.

Fuente: Anexo 1.

La Figura 14 muestra los casos de uso del proceso de compras en línea, es decir las actividades que se realiza para llevar a cabo un proceso, en este caso se identificaron los casos de uso a partir del modelo del proceso de negocio de comercio electrónico diagramado en el Anexo 1. Las actividades de ingresar al sistema, realizar su registro, visualizar productos, buscar productos, agregar a carrito de compra los productos seleccionados, comprar estos productos y ver las compras realizadas; son actividades que usualmente hace el usuario en un entorno de comercio electrónico.

Para la identificación de los microservicios se tomó como referencias los casos de uso presentados, que representan los requerimientos funcionales de la aplicación web de comercio electrónico.

En la Tabla 1 se identifica los microservicios de usuarios, productos y ventas a través de las funcionalidades expresadas en tareas específicas que realiza la aplicación web, en el que se consideró como criterios de selección el valor de la tarea, la escalabilidad y entidad de datos.

Tabla 1

Microservicios identificados del proceso de compras en línea.

Microservicios	Tareas (Responsabilidad individual)	Valor de la tarea	Escalabilidad	Entidad de datos
Usuarios	Registro	1	1	Usuarios
	Inicio de Sesión	2	2	
	Confirmar Email	1	1	
	Ver Perfil	1	1	
Productos	Ver Lista productos	2	3	Productos
	Ver Detalle de Producto	2	3	
	Buscar productos	2	3	
Ventas	Agregar a carrito de compra	3	3	Ventas
	Comprar productos	3	3	
	Cantidad de Ítems en carrito	2	2	

Fuente: Anexo 1.

4.1.3 Discusión

Existen métodos y guías para la definición de la arquitectura, muchos de los cuales se focalizan en los requisitos funcionales. Sin embargo en la presente investigación se consideró lo que menciona el autor Bass (2012) de que es posible crear una arquitectura basada en las necesidades de atributos de calidad por lo que para la obtención de los requerimientos se consideró los atributos de calidad de eficiencia, rendimiento, disponibilidad, escalabilidad y seguridad, sin embargo se consideró los requisitos funcionales para la identificación de los microservicios para lo cual los casos de uso brindó una mayor abstracción de los microservicios candidatos y considerando las recomendaciones de Kharbuja (2016) de los criterios que influyen en la identificación del microservicio que son la responsabilidad individual, la escalabilidad y el valor de la tarea, en la investigación se aporta otro criterios que es la entidad de datos.

4.2 RESULTADOS CONFORME AL OBJETIVO ESPECÍFICO 2

4.2.1 Diseño de la composición de microservicios a nivel de servicios

Considerando el modelo del proceso de comercio electrónico detallado en el Anexo 1, se ha seleccionado siete tareas principales para la composición del servicio de compras online, es decir la aplicación Web de comercio electrónico.

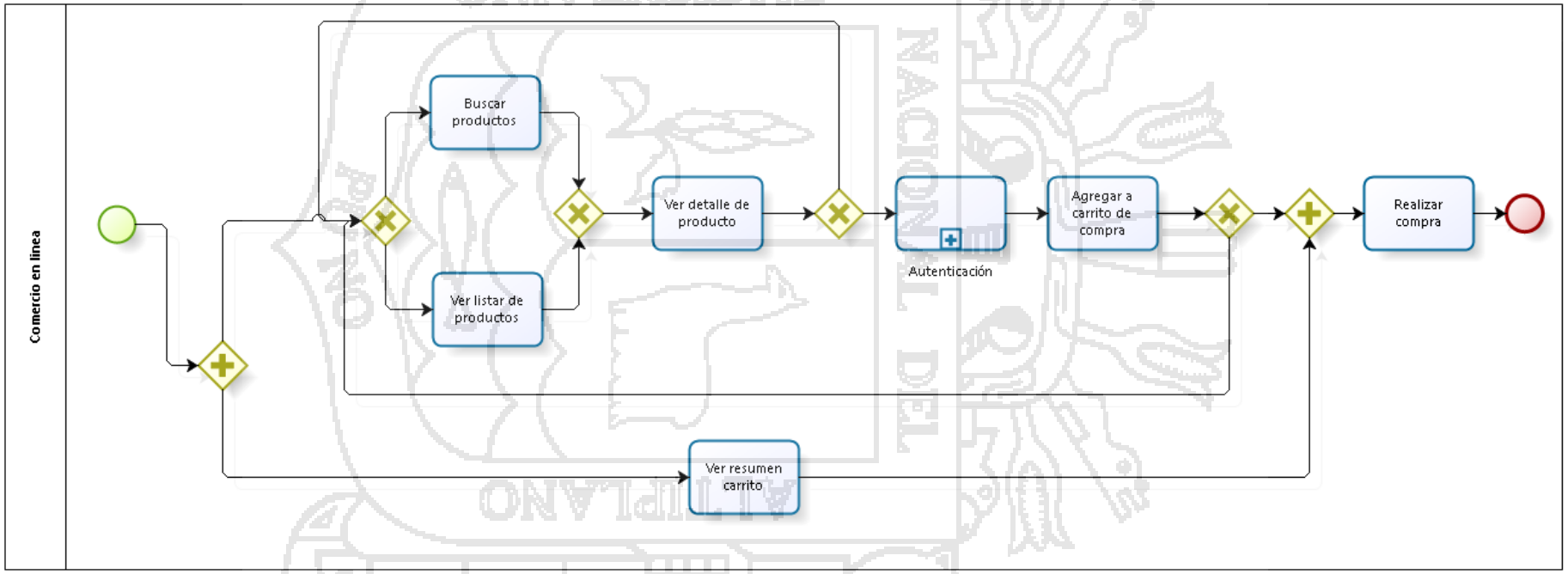


Figura 15. Modelo BPMN de la composición para la aplicación Web de comercio electrónico.

Fuente: Anexo 1.

La Figura 15 muestra, el diseño basado en modelo BPMN de la lógica de negocio de comercio electrónico, considerando las tareas principales anteriormente seleccionados; donde se describe el flujo de las tareas y actividades necesarias para brindar el servicio.

En la Tabla 2 se muestra los microservicios seleccionados con sus tareas respectivas según el tipo de petición y su *endpoint*, que son las rutas finales del recurso; que fueron utilizados para la construcción de la API; también se ha asignado un código a cada tarea para poder realizar su composición en forma algebraica.

Tabla 2

Microservicios y tareas del modelo propuesto.

Microservicios	**Tareas**	**Código**	**Tipo de petición**	**Rutas finales**
Usuarios	Inicio de sesión	AS1	POST	/inicioSesion
Productos	Ver lista productos	CL1	GET	/productos
	Ver detalle de producto	CP2	GET	/productos/detalle/{{n}}
	Buscar productos	CB3	POST	/productos/búsqueda
Ventas	Agregar a carrito de compra	VA1	POST	/ventas/carrito
	Comprar productos	VC2	POST	/ventas/checkout
	Cantidad de ítems en carrito	VR3	GET	/ventas/carrito

Fuente: Tabla 1, Anexo 1.

El álgebra de composición del servicio de comercio electrónico está dado por:

$$C_{co} = \{[(CB3 \diamond CL1) \odot CP2]^n \odot (AS1 \odot VA1)^m \odot VC2\} /\!/ VR3$$

Donde:

$/\!/$: *Invocación paralela.*

$\diamond$: *Invocación alternativa.*

$\odot$: *Invocación secuencial.*

n, m: *iteraciones.*

$m <= n$.

En la fórmula de composición del servicio de compras online se identifica peticiones al servicio en forma paralela, alternativa, secuencial y por iteraciones, que sirvieron de base para las pruebas de la composición de microservicios de manera cuantificable objetiva; donde el servicio de encontrar un producto con el buscador del catálogo y los servicios seleccionar producto de la lista de productos por categoría son actividades alternativas. Seguidamente el usuario visualiza el detalle del producto seleccionado, éste proceso lo puede realizar en forma cíclica hasta que el usuario decide agregar al carrito de compra, pero éste requiere ser autenticado por el sistema, en forma paralela se puede observar el resumen de carrito de compra; este proceso se puede realizar también de forma cíclica, lo que significa que el usuario puede agregar varios productos a su carrito de compra, finalmente se concretiza el servicio al realizar la compra.

4.2.2 Diseño de la composición de microservicios a nivel arquitectónico

La arquitectura lógica del modelo propuesto esta compuestos de cinco principales componentes lógicos, de los cuales cuatro son los microservicios que son: usuarios, productos, ventas y Frontend (interfaz de usuario), y un componente es el API Gateway, que es una puerta de interfaz que permite la comunicación con los microservicios. La Figura 16 muestra, la interrelación de los componentes lógicos de la arquitectura, en el que el componente API Gateway se encarga de integrar a los Microservicios y ser el intermediario entre el cliente y el servicio al que peticiona.

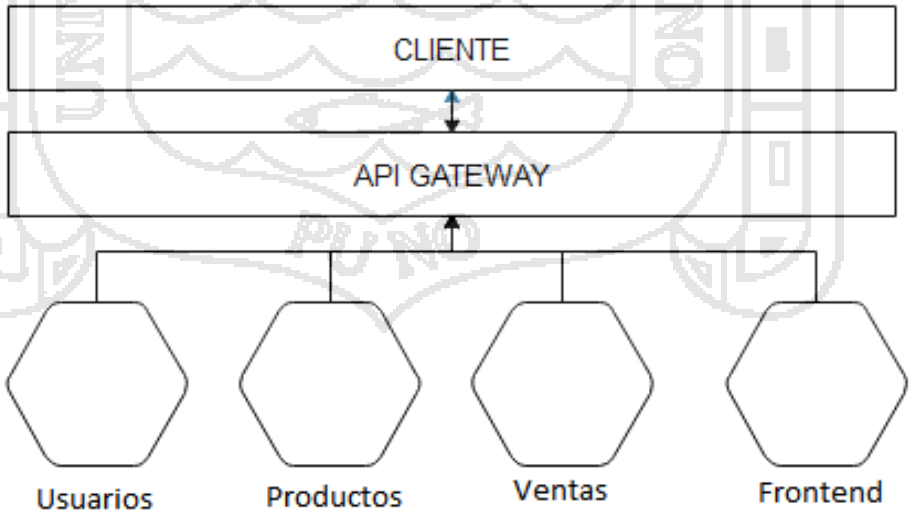


Figura 16. Arquitectura lógica del modelo composición de microservicios.
Fuente: Anexo1.

La arquitectura física del modelo propuesto presenta los componentes físicos: el cliente que puede ser un navegador Web, el clúster compuesto por 4 nodos de 3.75 GB de memoria RAM para cada microservicio, un componente PVC (Persistent Volume Claim) para la persistencia de la base de datos, un API Gateway, que utiliza el servidor proxy NGINX, encargado del descubrimiento de los microservicios por DNS o HTTP.

La Figura 16 muestra la relación de los componentes físicos y virtuales de la arquitectura del modelo propuesto. Se ilustra también que la comunicación de los Microservicios entre sí, se realiza a través de interfaces de programación de aplicaciones (API), éstas APIs fueron expuestas como endpoints de tipo REST.

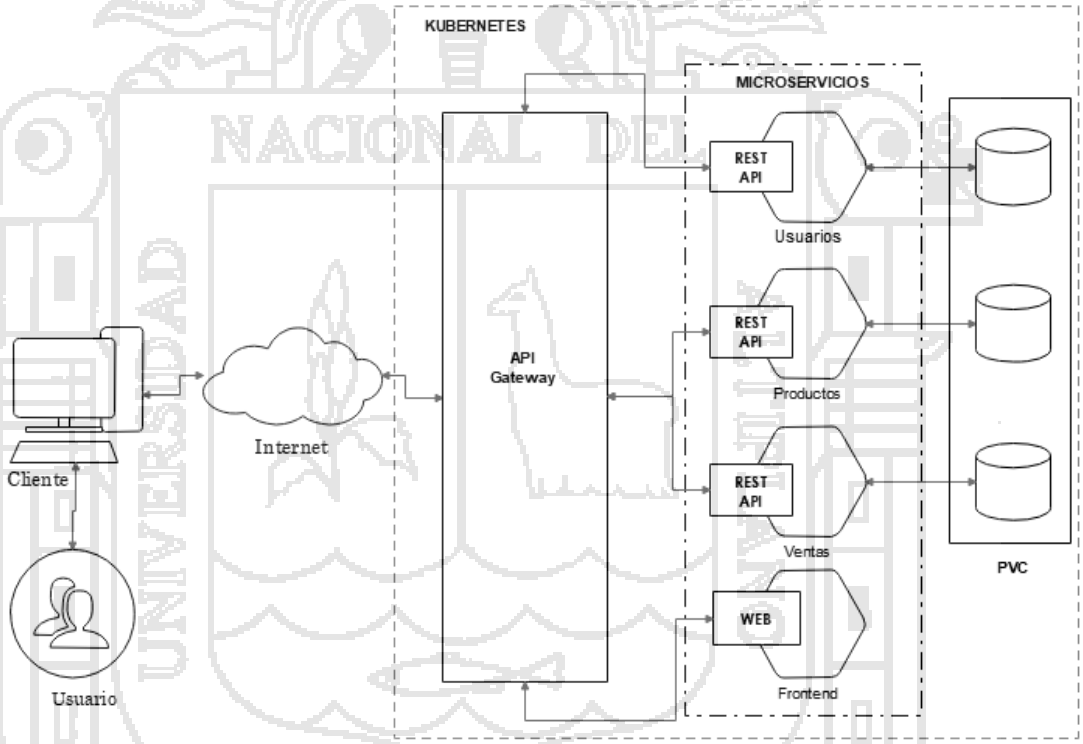


Figura 17. Arquitectura física del modelo de composición de Microservicios.
Fuente: Anexo 1.

4.2.3 Diseño de la seguridad de acceso a los recursos del microservicios

Para la seguridad de acceso a los microservicios se utilizó JWT, que es un estándar abierto basado en JSON, para crear tokens de acceso para pasar la identidad de los usuarios autenticados entre los diferentes microservicios.

La Figura 18 muestra la comunicación segura entre los microservicios donde el microservicio usuarios autentica y devuelve el token al cliente, y con ese token tanto el microservicio usuarios, productos y ventas pueden identificar al usuario que peticiona los servicios.

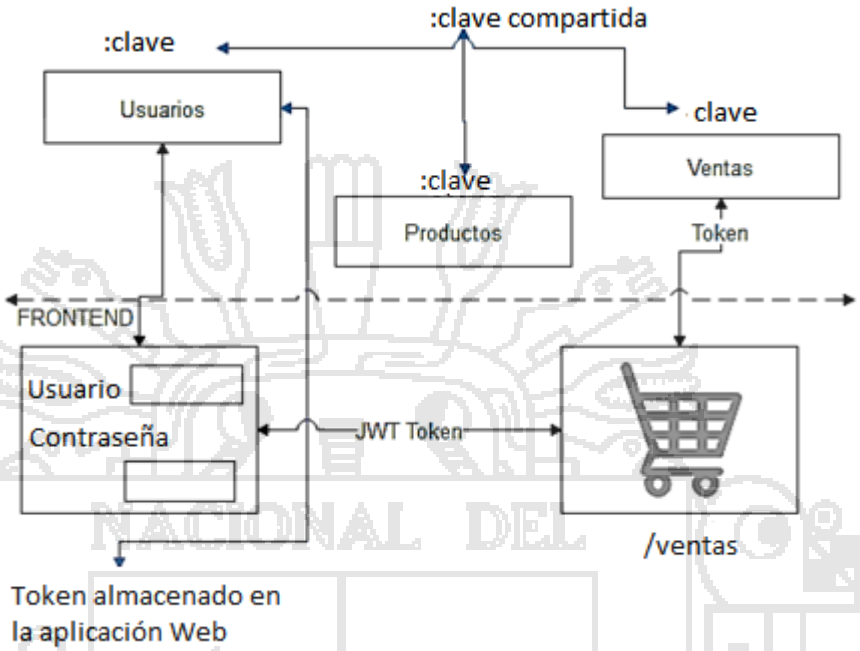


Figura 18. Comunicación segura entre Microservicios mediante Token.
Fuente: Anexo 3.

4.2.4 Discusión

Bellido (2015) para el diseño de identificación servicios SOA hace uso del modelo BPMN, sin embargo Kharbuja (2016) hace uso de los casos de uso para la identificación de los microservicios. En la presente investigación se hace uso de ambas técnicas para la identificación de microservicios y además se utiliza el álgebra de composición de microservicios que recomienda Hamadi (2003). Por otro lado Fowler (2014) sostiene que la arquitectura basada en microservicio es un enfoque para desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos livianos, a menudo una API de recursos. La arquitectura SOA intentó definir un conjunto de estándares para la interoperabilidad como XML Y SOAP, sin embargo en la presente investigación se adoptó REST como protocolo de comunicación de los microservicios, este protocolo es uno de los cuáles se adapta mejor para la computación distribuida, sin embargo puede estar comprometido con problemas de seguridad, pero éste fue tratada haciendo uso de JWT.

4.3 RESULTADOS CONFORME AL OBJETIVO ESPECÍFICO 3

Para la implementación de la aplicación Web basado en la composición de microservicios, los microservicios usuarios, productos y ventas se implementaron en forma independiente haciendo uso del espacio de trabajo Echo y se codificó en el lenguaje de programación Golang en el lado del servidor. El microservicio Frontend se implementó con el entorno de trabajo Vue.js. Para la seguridad de la comunicación de los microservicios se utilizó el servicio de JWT.

4.3.1 Implementación de los microservicios

La implementación de los microservicio se hizo de forma independiente como si fuera una sola aplicación con estructura de carpetas y archivos organizados como se muestra en la Figura 19, donde se identifica la carpeta modelo (*models*) que contiene la lógica del negocio, es decir las entidades que se involucran en el negocio; la carpeta controladores (*controllers*) contiene las clases y métodos que se implementaron; el archivo *usuarios.rest*, contiene la API del microservicio usuarios, el archivo *apidoc.html* contiene la documentación de la API del microservicios y finalmente el archivo *usuarios.yaml* contiene la configuración para el despliegue del microservicio en la nube.

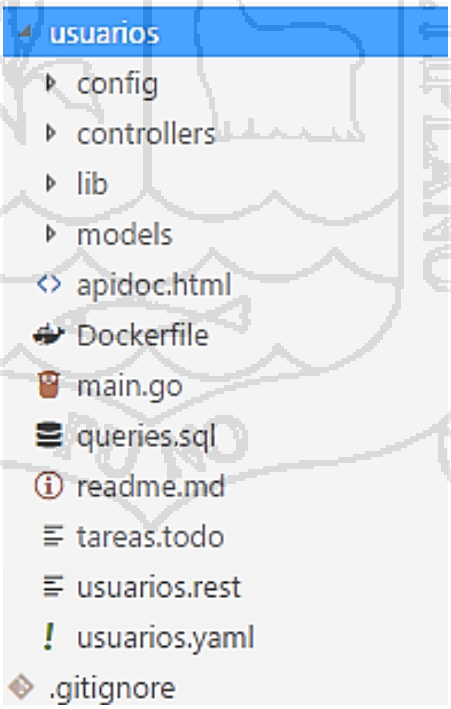


Figura 19. Directorio raíz del microservicio usuarios.

Fuente: Captura del código del microservicio usuarios.

En la Figura 20, se muestra la implementación de la función principal del microservicio usuarios, en el que se muestra que la codificación fue realizada en el lenguaje de programación Golang, ya que permitió que el código sea entendible y sencillo, además obtener sus ventajas que son en rendimiento nativo, rendimiento en tiempo real, concurrencia y paralelismo, escalabilidad, curva de aprendizaje y mejores prácticas y convenciones

```
package main
import (
    "fmt"
    "log"
    "net/http"
    "os/signal"
    "syscall"
    "github.com/braintree/manners"
    "github.com/doctorado/health"
    "github.com/doctorado/users/config"
    "github.com/doctorado/users/controllers"
    "github.com/doctorado/users/lib"
    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
    // Inicializamos con los parámetros de configuración
    dconfig.InitParams()
    errChan := make(chan error, 10)

      // Este servidor sirve para informar el estado del
microservicio a Kubernetes
    hmux := http.NewServeMux()
    hmux.HandleFunc("/healthz", health.HealthzHandler)
    hmux.HandleFunc("/readiness", health.ReadinessHandler)
                              hmux.HandleFunc("/healthz/status",
health.HealthzStatusHandler)
                             hmux.HandleFunc("/readiness/status",
health.ReadinessStatusHandler)
    healthServer := manners.NewServer()
    healthServer.Addr = *dconfig.HealthAddr
    healthServer.Handler = lib.LoggingHandler(hmux)


    go func() {
        errChan <- healthServer.ListenAndServe()
    }()
```

```
        e := echo.New()
        e.HideBanner = true

        e.Use(middleware.Logger())
        e.Use(middleware.Recover())
        e.Use(middleware.CORSWithConfig(middleware.CORSConfig{
            AllowOrigins: []string{"*"},
             AllowMethods: []string{echo.GET, echo.HEAD, echo.PUT,
    echo.PATCH, echo.POST, echo.DELETE},
        }))

        e.GET("/metrics", echo.WrapHandler(promhttp.Handler()))
        e.POST("/login", controllers.Login)
        e.POST("/users", controllers.CreateUser)
        e.GET("/users/confirm", controllers.GetConfirm)
        e.GET("/usersfake/:cantidad", controllers.FillFakeUsers)
        e.GET("/users/:id", controllers.GetUser)
        e.PUT("/users/:id", controllers.UpdateUser)
        e.DELETE("/users/:id", controllers.DeleteUser)
        r := e.Group("/admin")

        // Configuración del middleware con custom claims de JWT
        config := middleware.JWTConfig{
            Claims:     &lib.JwtCustomClaims{},
            SigningKey: []byte(*dconfig.Secret),
        }
        // Protección del grupo creado bajo la protección de JWT
        r.Use(middleware.JWTWithConfig(config))
        r.GET("/users", controllers.GetUsers)
        e.Logger.Fatal(e.Start(*dconfig.HttpAddr))
        signalChan := make(chan os.Signal, 1)
        signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)
        for {
            select {
            case err := <-errChan:
                if err != nil {
                    log.Fatal(err)
                }
            case s := <-signalChan:
                 log.Println(fmt.Sprintf("Captured %v. Exiting...",
    s))
                health.SetReadinessStatus(http.StatusServiceUnavaila
    ble)
              os.Exit(0)
            }
        }
    }
```

Figura 20. Código de la función principal del microservicio usuarios.

4.3.2 Despliegue de los microservicios

Los contenedores Docker ejecutan los microservicios bajo un *kernel* de Linux, para ello fue necesario compilar los ejecutables de cada microservicio en una computadora con sistema operativo Linux, también se pudo desde el subsistema de Linux de Windows 10, compilándolo de forma estática para evitar errores de dependencias de bibliotecas que el microservicio necesite en tiempo de ejecución.

El comando para compilar el microservicio, con golang, de forma estática es el siguiente:

go build –tags netgo –ldflags '-extldflags "-lm –lstdc++ -static"'

Luego se integró en una imagen Docker para que éste pueda ser utilizado en el clúster Kubernetes. La creación de la imagen Docker se realizó a través de la configuración de requerimientos tanto de archivos como del contenedor, utilizando un archivo llamado ***Dockerfile***, el cual incluye al microservicio compilado estáticamente, en este caso la Figura 21, muestra la configuración del microservicio de usuarios.

Dockerfile

```
1   FROM alpine:3.1
2   MAINTAINER Somebody
3   ADD usuarios /usr/bin/usuarios
4   ENTRYPOINT ["usuarios"]
5
```

Figura 21. Configuración Docker del microservicio usuarios.
Fuente: Anexo 7.

La generación de la imagen se realizó desde el terminal con el siguiente comando: **docker build -t gcr.io/doctorado-182201/usuarios**

La etiqueta asignada está compuesta de la dirección del repositorio de Google, llamado Container Registry seguido del identificador del proyecto, que en este caso es ***doctorado-182201***, finalmente el nombre de la imagen, la misma que se utilizó para invocarlos con Kubernetes.

La Figura 22 muestra el listado de imágenes disponibles de forma local en la máquina de trabajo que se obtuvo ejecutando el comando ***docker images***

```
$ docker build -t gcr.io/doctorado-182201/usuarios .
Sending build context to Docker daemon  58.89MB
Step 1/4 : FROM alpine:3.1
 ---> 00772ebf9244
Step 2/4 : MAINTAINER Somebody
 ---> Using cache
 ---> 175a8b794575
Step 3/4 : ADD users /usr/bin/users
 ---> Using cache
 ---> c50c83979dde
Step 4/4 : ENTRYPOINT users
 ---> Using cache
 ---> 1596a34b9692
Successfully built 1596a34b9692
Successfully tagged gcr.io/doctorado-182201/usuarios:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Wi
uild context will have '-rwxr-xr-x' permissions. It is recommended to double ch
tories.

$ docker images
REPOSITORY                                  TAG                 IMAGE ID
gcr.io/doctorado-182201/monolith            latest              ea510d80a561
gcr.io/doctorado-182201/products            latest              49433a1cbb89
gcr.io/doctorado-182201/users               latest              1596a34b9692
gcr.io/doctorado-182201/usuarios            latest              1596a34b9692
gcr.io/doctorado-182201/locust-tasks        latest              7140a2f05734
```

Figura 22. Lista de imágenes para ser desplegadas.

Fuente: Elaboración propia.

Seguidamente se subió la imagen,del microservicio usuarios generada, al repositorio de Google, Container Registry, utilizando el siguiente comando: **$ gcloud docker -- push gcr.io/doctorado-182201/usuarios**. De forma similar se realizaron para los demás microservicios, este proceso de despliegue se detalla en el Anexo 6. Finalmente en la Figura 23, se muestra los microservicios subidos.

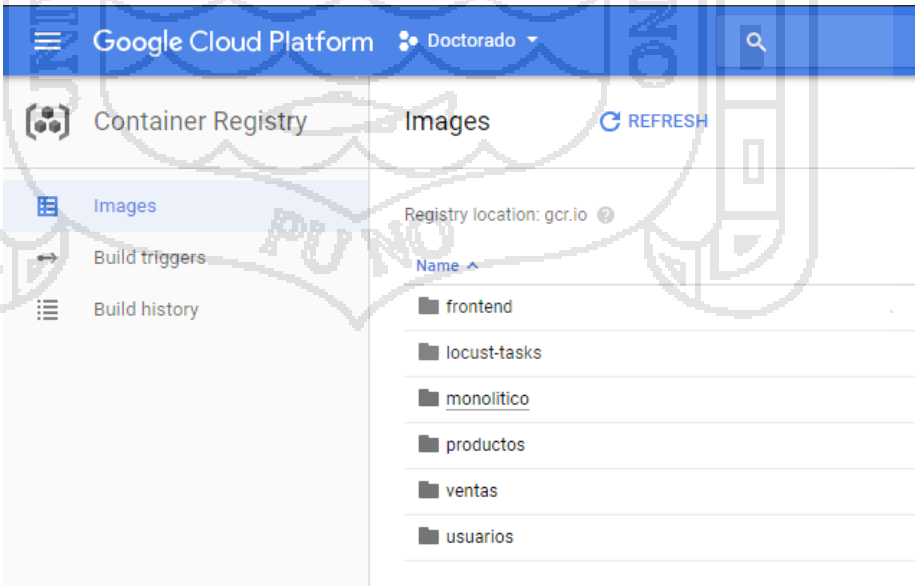


Figura 23. Repositorio Container Registry de los microservicios.

Fuente: Captura de las imágenes subidas a Google Cloud Plataform.

4.3.3 Evaluación de la calidad de los microservicios implementados

En la Tabla 3, se muestra los resultados de la evaluación de los atributos de calidad, que son: la granularidad, acoplamiento, cohesión, complejidad y reusabilidad, de los microservicios implementados y servicios implementados bajo la arquitectura monolítica. Esta evaluación se realizó después de implementar cada microservicio y se calculó utilizando las fórmulas de los atributos de calidad de las métricas de calidad de los microservicios propuestos por Kharbuja (2016).

Tabla 3

Comparación de los servicios y microservicios implementados respecto a sus atributos de calidad.

Atributos de calidad	Servicios	Microservicios
Granularidad	2.70	4.00
Acoplamiento	3.00	0.63
Cohesión	0.07	0.36
Complejidad	0.12	1.00
Reusabilidad	0.00	4.00

Fuente: Anexo 7.

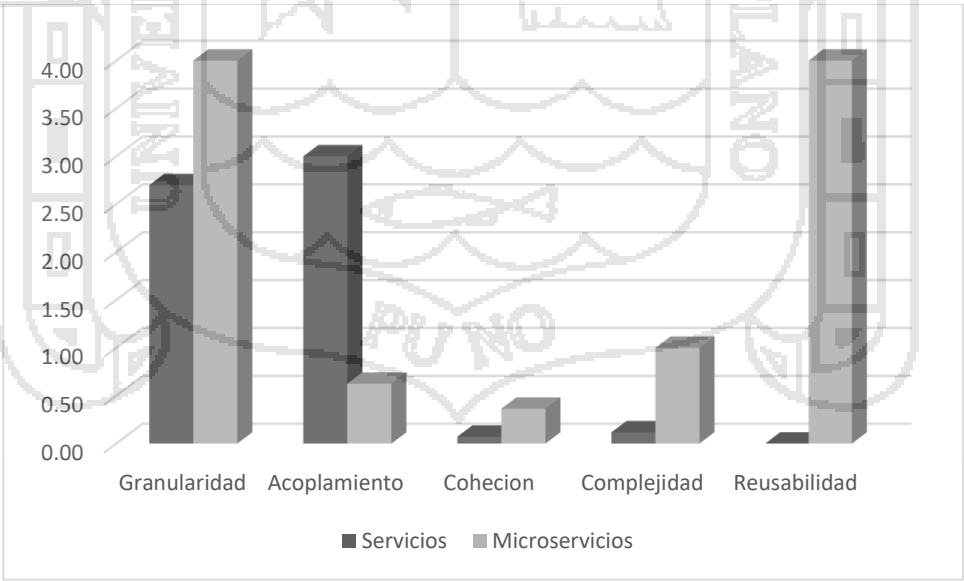


Figura 24. Comparación en atributos de calidad de los servicios y microservicios de la aplicación web de comercio electrónico.

Fuente: Tabla 4.

La Figura 24 muestra, la comparación de los resultados de la evaluación de los servicios y microservicios en los atributos de calidad que se describen a continuación:

La granularidad es un atributo de calidad que mide el tamaño adecuado de un microservicio; de esta se obtuvo una granularidad de 2.70 en los servicios de la arquitectura monolítica dado que esta trabaja sobre una sola aplicación, en comparación a los microservicios implementados, la cual subdivide las tareas en microservicios y de esta forma es posible tener servicios especializados en una determinada tarea.

El acoplamiento mide la dependencia entre servicios/operaciones; de la cual como se observa en la Tabla 3 se obtuvo 3.00 y 0.63 en los servicios de la arquitectura monolítica y arquitectura de microservicios respectivamente, lo cual implica que los microservicios implementados son altamente independientes de servicios/operaciones en comparación de la arquitectura monolítica, donde los servicios implementados se encuentra en una sola aplicación.

La complejidad está dado por la granularidad y el número de microservicios, de la cual se obtiene los valores 0.12 y 1.00 para los servicios de la arquitectura monolítica y los microservicios respectivamente, lo cual implica que la arquitectura de microservicios tiene mayor complejidad al ser este de mayor granularidad, lo cual conlleva a tener una cantidad considerable de microservicios y su gestión se torna en una actividad de gran complejidad para conocer las tareas que realizan determinados microservicios que están en determinados servidores.

La cohesión mide la relación de una función implementada y su propósito, en la evaluación de este atributo se obtuvo que los servicios de la arquitectura monolítica tienen baja cohesión, 0.07, mientras que los microservicios tienen una alta cohesión 0.36, lo que indica que la implementación de un microservicio facilita el entendimiento de una clase o método y éstos puedan ser reutilizados.

La reusabilidad mide la capacidad de reutilización de un determinado microservicio, donde se obtuvo los valores 0.00 y 4.00 para los servicios de la arquitectura monolítica y los microservicio respectivamente, en la cual se observa que los servicios de la arquitectura monolítica no puede ser reutilizada al ser una aplicación integra, mientras que los microservicios al ser tareas especializadas, éstas tienen un

indicador de reusabilidad elevado, lo cual implica que estos microservicios pueden ser utilizados independientemente para la implementación de otras aplicaciones de negocio.

4.3.4 Discusión

Los Microservicios implementados se basan en las capacidades empresariales, es decir en los procesos de negocio que se analizó y diseño. A diferencia de la arquitectura monolítica, cada microservicio se implementó de forma independiente con un distinto equipo de desarrollo basado en las recomendaciones de Sánchez (2014).

Se observó que al trabajar de forma independiente cada microservicio y con su propio equipo de desarrollo, la culminación del desarrollo de cada microservicio fue en menor tiempo, sin embargo se tomó un tiempo adicional para la integración de estos microservicios.

4.4 RESULTADOS CONFORME AL OBJETIVO ESPECÍFICO 4

El contexto de las pruebas de carga se llevó dentro de un clúster de Kubernetes para evaluar el modelo de composición de microservicios de la aplicación Web de comercio electrónico implementada.

El objetivo de la prueba fue evaluar el modelo propuesto bajo carga, comparándolo con el modelo monolítico en los indicadores de tiempo de respuesta, disponibilidad y rendimiento.

En la fase de diseño de las pruebas de carga se codificó, en lenguaje de programación Python, las pruebas de cada funcionalidad de la composición de los microservicios de la implementación de la aplicación Web de comercio electrónico. Estas pruebas se codificaron en el archivo *pruebas.py* que se muestra en el Anexo 4.

En la fase de ejecución de las pruebas de carga se realizó la simulación utilizando la herramienta tecnológica Locust, su configuración se muestra en el Anexo 3, donde se consideró 500 usuarios, con una tasa de eclosión de 10 usuarios, que realizan peticiones por segundo a cada funcionalidad de la composición de los microservicios de la aplicación Web de comercio electrónico implementada. La simulación de la prueba de carga se desarrolló durante un tiempo considerable para alcanzar las 20 peticiones por segundo.

Los datos del comportamiento de este proceso de prueba de carga, se presenta en en el Anexo 6.

En la fase de análisis y resultados de la prueba de carga, se enfocaron en los indicadores de tiempo de respuesta, disponibilidad y rendimiento que se presentan a continuación.

4.4.1 Tiempo de respuesta

En la Tabla 4 se muestra los resultados en el indicador de tiempo de respuesta expresado en milisegundos del modelo propuesto de composición de microservicio y e l modelo monolítico, en el cual se considera el tiempo de respuesta promedio de los diferentes microservicios y el tiempo promedio de respuesta del modelo monolítico.

Tabla 4

Tiempo de respuesta del modelo de composición de Microservicios propuesto y la arquitectura monolítica.

	Tiempo de respuesta(ms)		
peticiones/segundo	Monolítico	Modelo propuesto	% de mejora
2	498	190	62
4	598	225	62
6	1700	299	82
8	1692	376	78
10	2212	393	82
12	2125	451	79
14	2403	444	82
16	2512	512	80
18	2630	681	74
20	3021	750	75

Fuente: Anexo 6.

En la Figura 25 se puede observar una mejora del 75.17% del modelo propuesto de composición de Microservicios frente al modelo monolítico, en cuanto al tiempo de respuesta cuando se realiza 20 peticiones por segundo, se observa que las peticiones que son realizadas en una arquitectura de Microservicios es en una computación distribuida, pero se delimita a 20 peticiones por minuto debido al ancho que se nos

otorga al lado cliente y la simulación se realizó desde una máquina con los servidores, donde los mismos al estar clúster permite realizar una respuesta en un tiempo menor al modelo monolítico debido que al ser independiente y de funcionalidad específica, tiene menos tareas que el modelo monolítico. Además que éstos microservicios se encuentran en su propio host, es decir se ejecutan en su propio proceso.

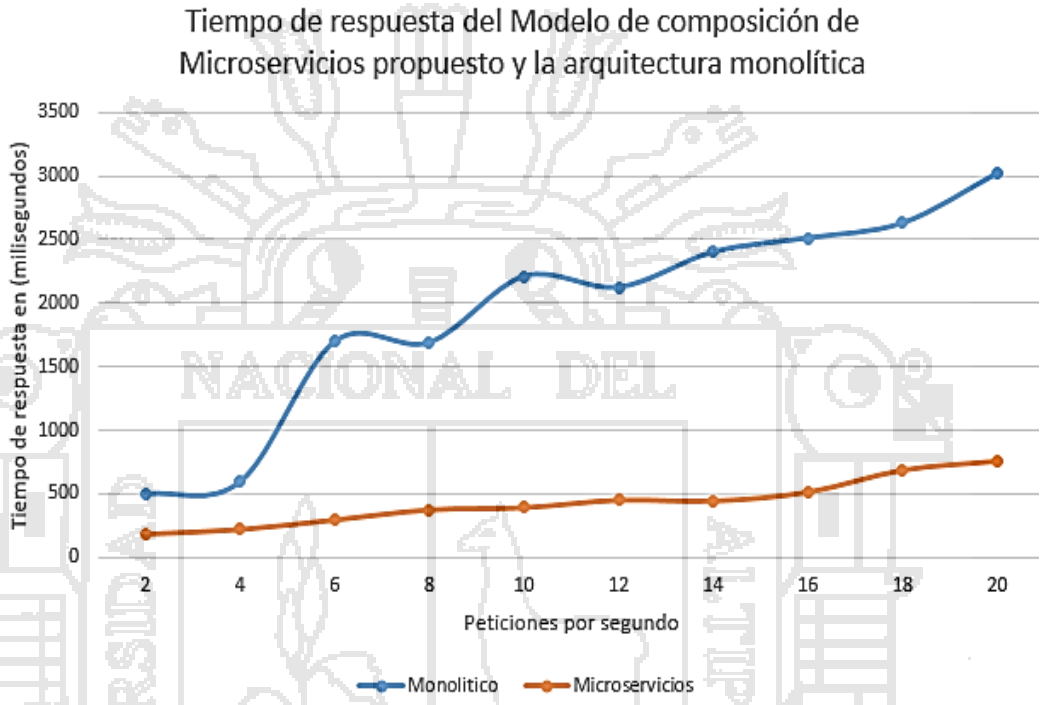


Figura 25. Tiempo de respuesta del modelo de composición de Microservicios propuesto y la arquitectura monolítica.
Fuente: Anexo 6.

4.4.2 Disponibilidad

Se presenta el resumen indicador disponibilidad, referida a la capacidad de la aplicación web para brindar un servicio 24/7 a los usuarios que la accedan, es decir un servicio que este las 24 horas del día los 7 días de la semana en las distintas zonas horarias.

En la Tabla 5, se muestra el resumen indicador disponibilidad del modelo de composición de microservicios y del modelo monolítico, donde la disponibilidad fue calculada a través del número de peticiones a la aplicación web y los errores que presenta durante éstas peticiones. La ejecución de las pruebas re realizó de la aplicación web de comercio electrónico desarrollada por ambos modelos.

Tabla 5

Disponibilidad del modelo de composición de Microservicios propuesto y la arquitectura monolítica.

peticiones/ segundo	Disponibilidad (%)		% de mejora
	Monolítico	Modelo propuesto	
2	100	100	0
4	100	100	0
6	95	100	5
8	79	100	21
10	68	96	28
12	70	93	23
14	57	86	29
16	52	87	35
18	43	86	43
20	37	81	44

Fuente: Anexo 4.

En la Figura 26 en la cual se puede observar que ambos modelos responden a 100% de disponibilidad inicialmente, pero cuando se realiza de 6 peticiones por segundo a más se va observando como baja la disponibilidad del modelo monolítico la cual llega hasta perder el 63% de la disponibilidad en este modelo al realizar 20 peticiones por segundo, y así mismo se observa que en el modelo microservicios solo llega a perder 20% de las peticiones al realizar 20 peticiones por segundo y acuerdo a la curva que se observa que el modelo de microservicios muestra mayor disponibilidad a diferencia del modelo monolítico; lo que significa que la aplicación web de comercio electrónico brinda el servicio a sus funcionalidades en forma continua, y si existe algún error que pueda surgir al peticionar un servicio, éste servicio se encuentra replicado en otro servidor, garantizando de esta forma la alta disponibilidad de la aplicación web, a diferencia del modelo monolítico, al surgir un error se pierde disponibilidad debido a que se tiene que solucionar los problemas del servidor y esto interrumpe su servicio de funcionalidades.

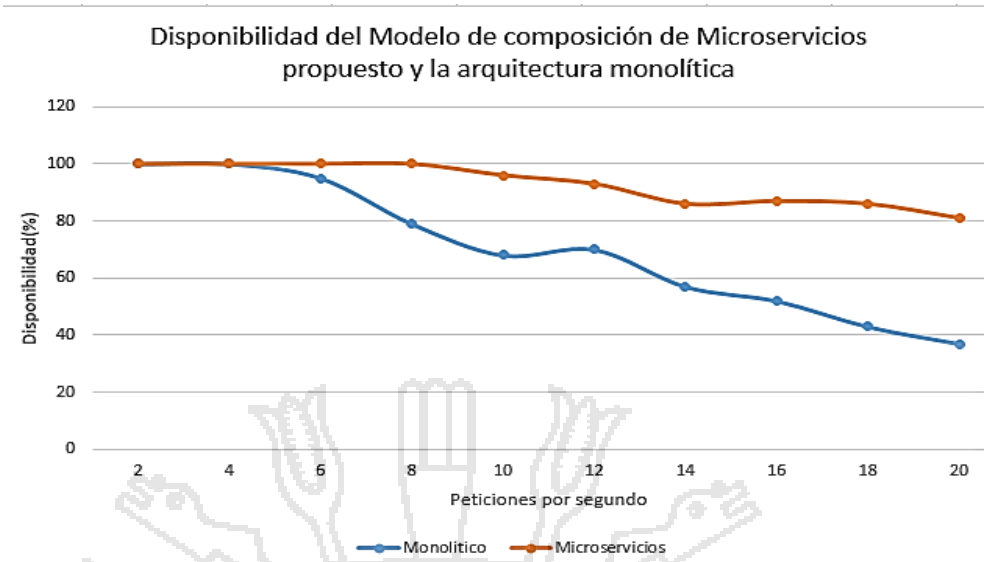


Figura 26. Disponibilidad del modelo de composición de microservicios propuesto y la arquitectura monolítica.

Fuente: Anexo 6.

4.4.3 Rendimiento

En la Tabla 6 muestra, el resumen del rendimiento de los modelo de composición de microservicio propuesto y del modelo monolítico, donde el rendimiento fue calculada a través de una estimación de tiempo de 60 segundos para poder obtener los datos expuestos, la prueba de carga fue ejecutada a través de peticiones por segundos.

Tabla 6

Rendimiento del modelo de composición de microservicios propuesto y la arquitectura monolítica.

	Rendimiento(peticiones/minuto)		
peticiones/segundo	**Monolítico**	**Modelo propuesto**	**% de mejora**
2	120	120	0
4	240	240	0
6	350	360	3
8	408	480	18
10	510	600	18
12	503	698	39
14	495	814	64
16	422	820	94
18	400	743	86
20	252	740	194

Fuente: Anexo 6.

La Figura 27 ilustra que durante el tiempo estimado, se obtuvo un rendimiento óptimo en ambos modelos, pero a partir de 6 peticiones por segundo el modelo monolítico se observa que baja su rendimiento el cual se da a razón de su arquitectura y es a partir de este punto que baja su rendimiento de este modelo; en cuanto el modelo de composición de microservicios se observa que a partir de 12 peticiones por segundos baja su rendimiento donde una de las limitantes que se observa es el ancho de banda dado que en el tema de procesamiento y memoria no se observa saturación alguna.

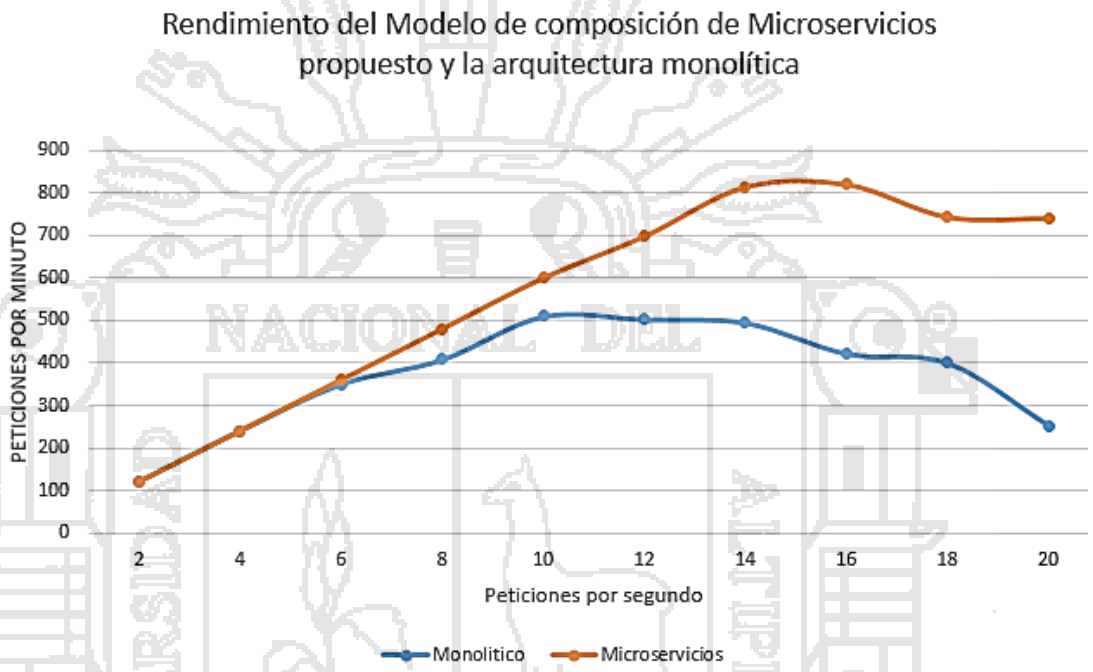


Figura 27. Rendimiento del modelo de composición de microservicios propuesto y la arquitectura monolítica.
Fuente: Anexo 6.

4.4.4 Discusión

En la presente investigación se realizó pruebas de carga para la evaluación del modelo propuesto de composición de microservicios para la implementación de una aplicación Web de comercio electrónico.

El autor Jiang (2015) describe dos enfoques para realizar las pruebas de cargas las realistas que tiene como objetivo diseñar una prueba de carga, que se parece mucho al uso esperado en el campo y el diseño de pruebas de carga inductoras de fallas apunta a diseñar se realiza las pruebas de los modelos planteados, sin embargo se consideró convenientemente el primer enfoque.

En el análisis de los resultados obtenidos de la prueba de carga se encuentra que el modelo de microservicios es más eficiente que el modelo monolítico, con respecto a los indicadores de tiempo de respuesta, disponibilidad y rendimiento, a razón de que el modelo microservicios se basa en una arquitectura distribuida, en la cual se divide el modelo en las funciones más básicas las cuales son determinadas como microservicios y solo se dedican a esta función, para lo cual cada microservicio cuenta con un servidor independiente por lo cual tiene un mejor tiempo de respuesta, mayor rendimiento y disponibilidad que el modelo monolítico, ya que en este modelo todas las funciones se establecieron solo en un servidor, sin embargo este modelo cuenta con un mayor tiempo de respuesta, menor rendimiento y disponibilidad en comparación al modelo de microservicios, así mismo se observa que este modelo de microservicios tiene mayor facilidad en el momento de la escalabilidad, esto a razón que se puede agregar nuevas funcionalidades o dar mantenimiento a las mismas debido a los despliegues que se puede realizar de cada microservicio.

4.5 PRUEBA DE HIPÓTESIS

Para la prueba de hipótesis con respecto al modelo de composición de microservicios, para la implementación de una aplicación Web de comercio electrónico, se ha utilizado como método de prueba de hipótesis la denominada prueba t-student, proceso que se ha realizado utilizando SPSS, para lo cual se plantearon las siguientes hipótesis:

Con respecto al tiempo de respuesta se ha formulado las siguientes hipótesis estadísticas:

H_O: El tiempo de respuesta de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es mayor o igual que el modelo monolítico.

H_a: El tiempo de respuesta de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes, es menor al modelo monolítico.

En la Tabla 7, se muestra que Sig = 0.000, es decir Sig <0.05 entonces se rechaza la hipótesis nula y se acepta la hipótesis alterna que afirma que el tiempo de respuesta de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es menor que el implementado con el modelo monolítico.

Tabla 7

Prueba de muestras independientes con respecto al tiempo de respuesta.

Prueba de muestras independientes						
		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias		
		F	Sig.	t	gl	Sig. (bilateral)
Tiempo de respuesta	Se asumen varianzas iguales	11.340	0.003	5.570	18	0.000
	No se asumen varianzas iguales			5.570	9.838	0.000

Fuente: Anexo 6.

Con respecto disponibilidad se ha formulado las siguientes hipótesis estadísticas:

H_O: La disponibilidad de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es menor o igual que el modelo monolítico.

H_a: La disponibilidad de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es mayor al modelo monolítico.

Tabla 8

Prueba de muestras independientes con respecto a la disponibilidad.

Prueba de muestras independientes						
		Prueba de Levene de igualdad de		prueba t para la igualdad de medias		
		F	Sig.	t	gl	Sig. (bilateral)
Disponibilidad	Se asumen varianzas iguales	9,885	,006	-2,970	18	,008
	No se asumen varianzas iguales			-2,970	10,785	,013

Fuente: Anexo 6.

Con respecto al rendimiento se ha formulado las siguientes hipótesis estadísticas:

H_O: El rendimiento de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes, es menor o igual que el modelo monolítico.

H_a: El rendimiento de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes, es mayor al modelo monolítico.

Tabla 9

Prueba de muestras independientes con respecto al rendimiento.

Prueba de muestras independientes

		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias		
		F	Sig.	t	gl	Sig. (bilateral)
Rendimiento	Se asumen varianzas iguales	5.983	0.025	-2.152	18	0.045
	No se asumen varianzas iguales			-2.152	13.535	0.050

Fuente: Anexo 6.

En la Tabla 09, se muestra que Sig = 0.045, es decir Sig <0.05 entonces la regla de la decisión es rechaza la hipótesis nula y se acepta la hipótesis alterna, que afirma que el rendimiento de la aplicación Web de comercio electrónico basado en el modelo de composición de microservicios utilizando Kubernetes es mayor que el implementado bajo el modelo monolítico.

CONCLUSIONES

- El modelo de composición de microservicios, basado en la arquitectura de microservicios, para la implementación de una aplicación Web de comercio electrónico utilizando Kubernetes funciona significativamente en comparación con un modelo monolítico en un 104% con respecto a los indicadores de rendimiento, disponibilidad y tiempo de respuesta.

- El análisis de los requerimientos arquitectónicos permitió comprender cómo la aplicación Web de comercio electrónico debe funcionar significativamente, para lo cual se identificó cinco requerimientos que son la alta disponibilidad, seguridad, rendimiento, escalabilidad, eficiencia y rendimiento. Los requerimientos funcionales del sistema se identificaron a partir del mapa de proceso del negocio de comercio electrónico a través del modelo BPMN.

- El diseño del modelo propuesto se realizó a través del diseño de la arquitectura lógica y física, identificándose cuatro microservicios que son productos, ventas, usuarios y Frontend, estos microservicios fueron orquestados a través de una API Gateway, que actuó como una puerta de comunicación entre los diferentes microservicios y el orquestador.

- La implementación del modelo de composición de microservicios para la aplicación Web de comercio electrónico facilitó la implementación de cada microservicio, al poder ser implementado en forma independiente con libertad en el uso de tecnologías; para lo cual se utilizó el lenguaje de programación Golang, para la implementación en el lado del servidor y el marco de trabajo Vue, que facilitó la construcción del lado del cliente; así mismo se utilizó el servicio de autenticación JWT, que permitió la seguridad en el acceso a los microservicios; y la herramienta tecnológica Kubernetes, que facilitó la orquestación de los contenedores, donde fueron desplegados los microservicios.

- La evaluación del modelo de composición de microservicios para una aplicación Web de comercio electrónico a través de pruebas de carga automatizada, permitió validar el comportamiento del modelo de composición de microservicios propuesto al ser comparado con un modelo monolítico.

RECOMENDACIONES

- Se recomienda el modelo propuesto de composición de servicios, basado en microservicios, frente a modelos arquitectónicos monolíticos, para el desarrollo de aplicaciones Web de comercio electrónico, ya que los usuarios esperan de ésta alta disponibilidad, mejor rendimiento y mayor tiempo de respuesta.
- Se recomienda en la fase de análisis de la metodología XP, enfatizar la identificación de los requerimientos arquitectónicos, ya que éstos garantizan la funcionalidad de calidad del sistema software.
- Se recomienda enfatizar en la fase de diseño en la metodología XP, el diseño de la arquitectura lógica y física a través de diagramas para la abstracción del comportamiento de los componentes que conformarán un sistema software. Así mismo se recomienda el modelo BPMN y el álgebra para el diseño de composición de servicios.
- Se recomienda la implementación de microservicios en forma independiente y paralela validadas a través del uso de las métricas de calidad en los atributos de calidad de granularidad, cohesión, complejidad, reusabilidad y acoplamiento.

BIBLIOGRAFÍA

Aderaldo, C. M., Mendonça, N. C., Pahl, C., & Jamshidi, P. (2017). Benchmark Requirements for Microservices Architecture Research. *IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), At Buenos Aires, Argentina*, (March), Microservices have recently emerged as a new ar-c. https://doi.org/10.1109/ECASE.2017..4

Alpers, S., Becker, C., Oberweis, A., & Schuster, T. (2015). Microservice based tool support for business process modelling. In *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOCW 2015*. https://doi.org/10.1109/EDOCW.2015.32

Bakshi, K. (2017). Microservices-based software architecture and approaches. In *IEEE Aerospace Conference Proceedings*. https://doi.org/10.1109/AERO.2017.7943959

Baryannis, G., Danylevych, O., Karastoyanova, D., Kritikos, K., Leitner, P., Rosenberg, F., & Wetzstein, B. (2010). Service Composition. *Service Research Challenges and Solutions for the Future Internet, 6500*, 55–84. https://doi.org/10.1007/978-3-642-17599-2_3

Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice. *Vasa, 2nd*, 1–426. https://doi.org/10.1024/0301-1526.32.1.54

Bellido, J. (2015). Dynamic Composition of REST services, 1–23.

Bonér, J. (2016). *Reactive Microservices Architecture*. Retrieved from https://www.lightbend.com/reactive-microservices-architecture?mkt_tok=eyJpIjoiTURJeVptVTRObUl5T0RVNCIsInQiOiIwNkdNb21tYWZpc2t6RXR2SzFmSlBXMlU4UVdvXC9DWCtmcWNReEw1Tm1tSlNtMHN4RUVJS0FqdjVEZ0djaFUzT3RSMXd3aE1oUkhVZm5PeW11S1lJRk9xNEJJRUY4dHRpdWl2V1FBXC9oTVhzP

Brüggemann, M. E., Vallon, R., Parlak, A., & Grechenig, T. (2014). Modelling microservices in email-marketing: Concepts, implementation and experiences.

ICSOFT-PT 2014 - Proceedings of the 9th International Conference on Software Paradigm Trends, 67–71. https://doi.org/10.5220/0005000800670071

Cabrera, M. (2016). Proyecto y formación en centros de trabajo 2º ASIR María Cabrera Gómez de la Torre.

CAPECE. (2016). Cámara peruana de comercio electrónico. Retrieved January 1, 2016, from http://www.cacepe.com

Claro, D. B., Albers, P., & Hao, J. (2006). Web services composition. *Semantic Web Services, Processes and Applications*, 195--225. https://doi.org/10.1007/978-0-387-34685-4_8

Cockcroft, A., Fellow, T., & October, B. V. (2016). Microservices Workshop : Why , what , and how to get there, (October).

Docker Inc. (2017). Understand images, containers, and storage drivers - Docker Documentation. Retrieved from https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices : yesterday , today , and tomorrow.

Eberhard, W. (2016). *Microservices: Flexible Software Architecture* (file:///C:).

Erl, T. (2005). *SOA: Principles of Service Design. 2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* https://doi.org/10.1109/SAINTW.2003.1210138

Fernández-Villamor, J. I., Iglesias, C. a., & Garijo, M. (2010). Microservices: lightweight service descriptions for REST architectural style. *Proc. of Second International Conference on Agents and Artificial Intelligence. Valencia, Spain*. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:MICROSERVICES+:+LIGHTWEIGHT+SERVICE+DESCRIPTIONS+FOR+REST+ARCHITECTURAL+STYLE#0

Fowler, M., & Lewis, J. (2014). Microservice Architecture. *Martinfowler.Com*. https://doi.org/10.1007/978-1-4842-1275-2_3

Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (pp. 91–101). https://doi.org/10.1145/336512.336537

Garriga, M., Mateos, C., Flores, A., Cechich, A., & Zunino, A. (2016). RESTful service composition at a glance: A survey. *Journal of Network and Computer Applications*. https://doi.org/10.1016/j.jnca.2015.11.020

Hamadi, R., & Benatallah, B. (2003). A Petri net-based model for web service composition. *Proceedings of the 14th Australasian Database Conference-Volume 17*, *12*(4), 191–200. https://doi.org/10.1007/s11741-008-0409-2

Haupt, F., Fischer, M., Karastoyanova, D., Leymann, F., & Vukojevic-Haupt, K. (2014). Service Composition for REST. In *Proceedings . IEEE 18th international Enterprise Distributed object computing conference* (Vol. 2014–Decem, pp. 110–119). https://doi.org/10.1109/EDOC.2014.24

INEI. (2016). Perú : Tecnología de Información y. *Encuesta Económica Anual 2015*.

Janakiram, M. (2017). *Designing Web-Scale Workloads with Microservices*. *Retrieved from http://info.dreamfactory.com/e1t/c/*VrbBnf5sqJrYN40RRD7j4q5s0/*W6zXrfc39C_SXVrFJ4R88Hykx0/5/f18dQhb0S5fw8XJ8g4W6vrn3M2YX2vsW7bf0LF83zdfyW74C1sh2zHMkcW6vp1yH3KfPW9W6b9BZn4DZpllW49vNQC4VbX0_W4Bs6cn4yym9nW2p0tdz5Q4vLRW4_GhWS4pBQZCW3sdnB02Gn9Xp*.

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using Docker technology. In *Conference Proceedings - IEEE SOUTHEASTCON* (Vol. 2016–July). https://doi.org/10.1109/SECON.2016.7506647

Jiang, Z. M. J. (2015). Load Testing Large-Scale Software Systems. In *Proceedings - International Conference on Software Engineering* (Vol. 2, pp. 955–956). https://doi.org/10.1109/ICSE.2015.304

Karunamurthy, R., Khendek, F., & Glitho, R. H. (2012). A novel architecture for Web service composition. *Journal of Network and Computer Applications*, *35*(2), 787–802. https://doi.org/10.1016/j.jnca.2011.11.012

Kharbuja, R. (2016). Desingning a Business Platform using Microservices. Retrieved from http://mediatum.ub.tum.de/doc/1285460/1285460.pdf

Letelier, P., & Penadés, M. C. (2006). Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP). *Técnica Administrativa*, *5*(26), 17. https://doi.org/1666-1680

Lewis, J., & Fowler, M. (2016). Microservices: A definition of this new architectural term. Retrieved from https://martinfowler.com/articles/microservices.html

Liu, D. (2013). Restful Service Composition, (October). Retrieved from http://homepage.usask.ca/~dol142/Files/rsc.pdf

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. In *Proceedings - 2017 IEEE 24th International Conference*

on Web Services, ICWS 2017 (pp. 524–531). https://doi.org/10.1109/ICWS.2017.61

Menasce, D. a. (2002). Load testing of Web sites. *IEEE Internet Computing*, *6*(4), 70–74. https://doi.org/10.1109/MIC.2002.1020328

Newman, S. (2015). *Building Microservices - Chapter 1, 4 and 11*. *Building Microservices*.

Oliveira, C., Lung, L. C., Netto, H., & Rech, L. (2017). Evaluating raft in docker on kubernetes. In *Advances in Intelligent Systems and Computing* (Vol. 539, pp. 123–130). https://doi.org/10.1007/978-3-319-48944-5_12

Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, (May), 137–146. https://doi.org/10.5220/0005785501370146

Pejman, E. ., Rastegari, Y. ., Majlesi Esfahani, P. ., & Salajegheh, A. . (2012). Web service composition methods: A survey. In *Lecture Notes in Engineering and Computer Science* (Vol. 2195, pp. 603–607). Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84867459385&partnerID=40&md5=5f92f96ca5bc81904b0af6a9a18e8a08

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, *36*(10), 46–52. https://doi.org/10.1109/MC.2003.1236471

Richards, M. (2015). *Microservices vs. Service-Oriented Architecture*. Retrieved from https://www.nginx.com/microservices-soa/

Richardson, C. (2014). Microservices: Decomposing Applications for Deployability and Scalability. Retrieved from http://www.infoq.com/articles/microservices-intro

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

Savchenko, D. I., Radchenko, G. I., & Taipale, O. (2015). Microservices validation: Mjolnirr platform case study. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015 - Proceedings* (pp. 235–240). https://doi.org/10.1109/MIPRO.2015.7160271

Steinacker, G. (2015). Von Monolithen und Microservices. Retrieved from http://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html#c9417

Strunk, A. (2010). QoS-aware service composition: A survey. In *Proceedings - 8th IEEE European Conference on Web Services, ECOWS 2010* (pp. 67–74). https://doi.org/10.1109/ECOWS.2010.16

Tarzey, B. (2015). What are containers and microservices? *Computerweekly.Com*,

(February 2016), 15–20. Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=113047319&site=ehost-live&scope=site

Ueda, T., Nakaike, T., & Ohara, M. (2016). Workload characterization for microservices. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016* (pp. 85–94). https://doi.org/10.1109/IISWC.2016.7581269

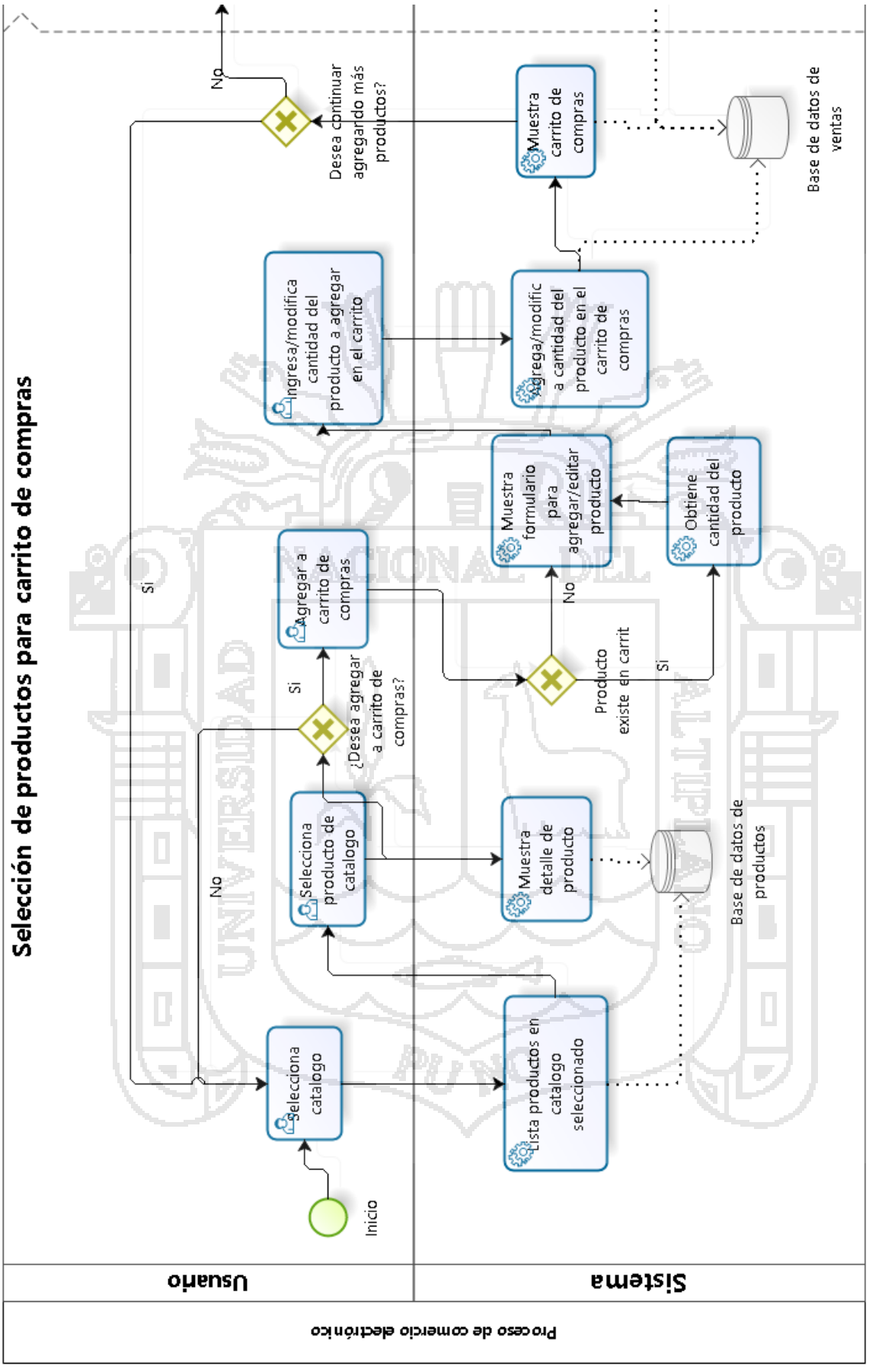
Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (pp. 583–590). IEEE. https://doi.org/10.1109/ColumbianCC.2015.7333476

Vivar, O. (2015). *Plataforma para la Anotación Semántica Automática de Servicios Web RESTful sobre un Bus de Servicios*. Cuenca-Ecuador.

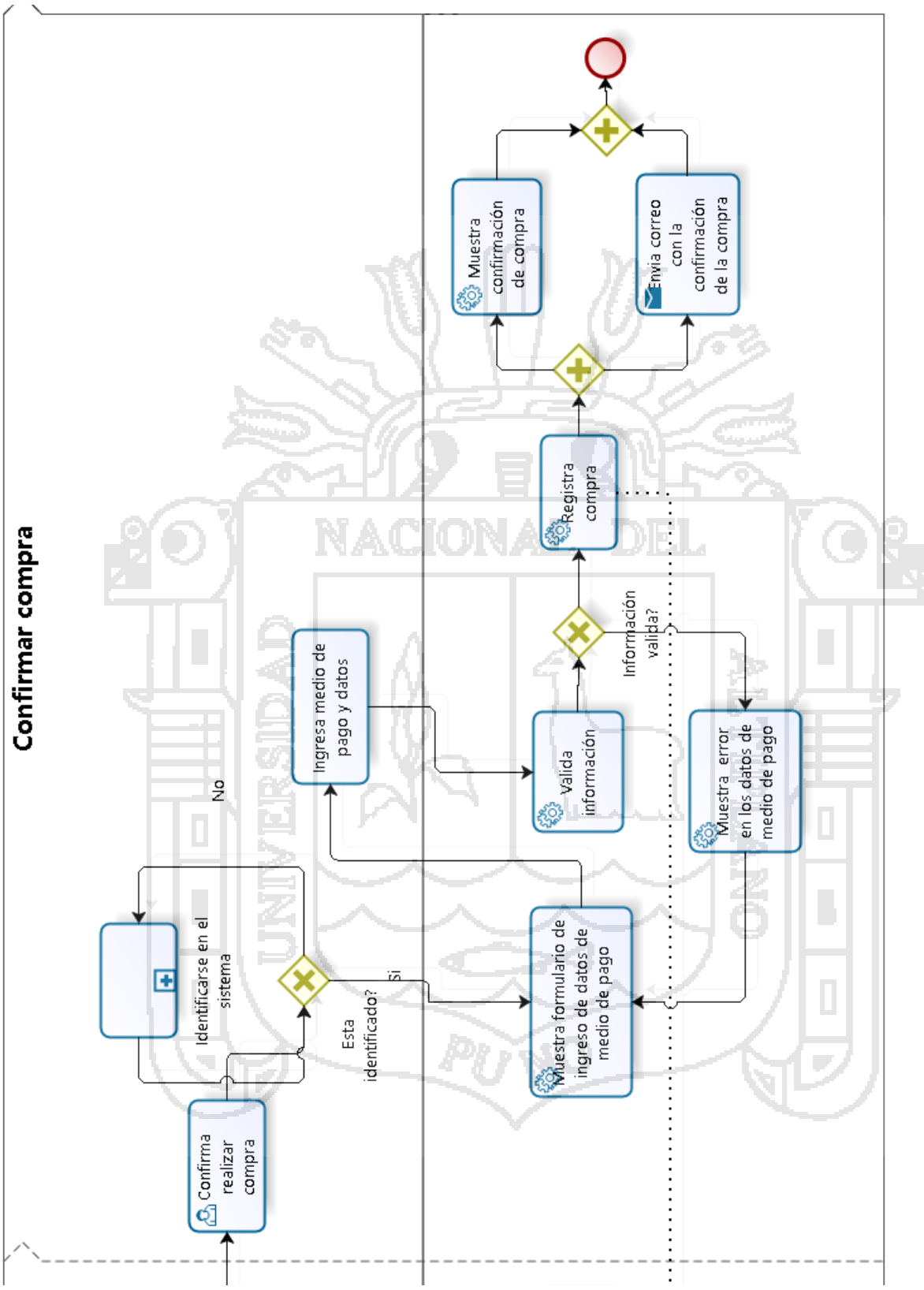
Webber, J. (2010). REST in practice. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 6285 LNCS, p. 7). https://doi.org/10.1007/978-3-642-15114-9_3

Xiao, Z., Wijegunaratne, I., & Qiang, X. (2017). Reflections on SOA and Microservices. In *Proceedings - 4th International Conference on Enterprise Systems: Advances in Enterprise Systems, ES 2016* (pp. 60–67). https://doi.org/10.1109/ES.2016.14

ANEXOS



Anexo 1. Diagrama del proceso de negocio de comercio electrónico.


Confirmar compra
Confirma realizar compra
Identificarse en el sistema
No
Esta identificado?
Si
Ingresa medio de pago y datos
Muestra formulario de ingreso de datos de medio de pago
Valida información
Información valida?
Muestra error en los datos de medio de pago
Registra compra
Muestra confirmación de compra
Envia correo con la confirmación de la compra

Anexo 2. Historias de usuario identificadas.

Número: 01	**Historia de Usuario**
Nombre de historia: Disponibilidad del sistema	
Como	Usuario
Quiero: acceder a la aplicación en cualquier momento dentro de las 24 horas de un día en cualquier momento del año.	
Para: realizar compras a través de la aplicación Web.	

Número: 02	**Historia de Usuario**
Nombre de historia: Seguridad en transacciones	
Como	Usuario
Quiero: que el sistema sea seguro	
Para: poder realizar transacciones de comercio electrónico.	

Número: 03	**Historia de Usuario**
Nombre de historia: Rapidez del sistema	
Como	Usuario
Quiero: que la aplicación Web sea rápida.	
Para: poder realizar una actividad de forma eficiente.	

Número: 04	**Historia de Usuario**
Nombre de historia: Nuevas funcionalidades	
Como	Usuario
Quiero: que la aplicación Web se actualice de forma continua con nuevas funcionalidades.	
Para: poder acceder las nuevas funcionalidades como promociones.	

Número: 05	**Historia de Usuario**
Nombre de historia: Atención concurrente	
Como	Usuario
Quiero: que el sistema en horas punta deberá tener la capacidad de atender a todos los usuarios.	
Para: poder acceder las nuevas funcionalidades como promociones.	

Anexo 3. Tareas de ingeniería identificadas.

TAREA DE INGENIERÍA	
Número de tarea: 01	**Número de historia:** 01
Nombre de tarea: Alta disponibilidad del sistema	
Tipo de tarea: No funcional	**Puntos estimados:** 5
Fecha inicio: 02/09/2017	**Fecha fin:** 06/07/2017
Programador responsable: Analista-programador 1	
Descripción: La aplicación Web deberá brindar una alta disponibilidad lo cual implica estar disponible 24/7.	

TAREA DE INGENIERÍA	
Número de tarea: 02	**Número de historia:** 02
Nombre de tarea: Seguridad del sistema	
Tipo de tarea: No funcional	**Puntos estimados:** 5
Fecha inicio: 09/09/2017	**Fecha fin:** 13/07/2017
Programador responsable: Analista-programador 2	
Descripción: La aplicación Web deberá brindar un servicio de autenticación a través de tokens para brindar la seguridad en el ingreso al sistema y acceso a los servicios. La comunicación de los servicios debe ser autenticada y encriptada utilizando certificados digitales.	

TAREA DE INGENIERÍA	
Número de tarea: 03	**Número de historia:** 03
Nombre de tarea: Escalabilidad	
Tipo de tarea: No funcional	**Puntos estimados:** 8
Fecha inicio: 16/09/2017	**Fecha fin:** 25/09/2017
Programador responsable: Analista-programador 2	
Descripción: El sistema deberá permitir agregar nuevos componentes al sistema y adaptarse a nuevas funcionalidades o cambios futuros.	

TAREA DE INGENIERÍA	
Número de tarea: 04	**Número de historia:** 04
Nombre de tarea: Eficiencia del sistema	
Tipo de tarea: No funcional	**Puntos estimados:** 6
Fecha inicio: 26/09/2017	**Fecha fin:** 02/10/2017
Programador responsable: Analista-programador 2	
Descripción: La aplicación Web deberá responder a las peticiones realizadas del usuario como máximo en 2 segundos para asegurar la eficiencia del sistema.	

TAREA DE INGENIERÍA	
Número de tarea: 05	**Número de historia:** 05
Nombre de tarea: Rendimiento	
Tipo de tarea: No funcional	**Puntos estimados:** 5
Fecha inicio: 16/09/2017	**Fecha fin:** 20/07/2017
Programador responsable: Analista-programador 1	
Descripción: El sistema deberá permitir la atención de las peticiones de usuarios concurrentes. (20 peticiones por segundo).	

Anexo 4. Codificación de las pruebas de carga al modelo de composición de microservicios.

pruebas.py

```
from locust import HttpLocust, TaskSet
from random import randint

def login(l):
    l.client.post("/login",
    {
        "username": "DeborahHall",
        "password": "password"
    })

def index(l):
    l.client.get("/")

def listaproductos(l):
    l.client.get("/products")

def search(l):
    l.client.get("/products/search?c=20&p=1&q=system")

def detalle(l):
    alea = randint(1, 50)
    l.client.get("/products/details/" + str(alea))

def checkout(l):
    l.client.request(method="POST",
    url="/esales/checkout",
    headers={
        "Authorization":                                    'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiRGVib3JhaEhhbGwiLCJ
yb2xlIjoiY2xpZW50ZSIsInV1aWQiOiIxIiwiZXhwIjoxNTExMzkzODk5fQ.v6s8moEB
CSMx_yaEuyTQ-tzoihT_S-f6g7Z3p8CmXbc'
    })
def addcart(l):
    alea = randint(1, 50)
    l.client.request(method="POST",
    url="/esales/cart",
    headers={
        "Authorization":                                    'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiRGVib3JhaEhhbGwiLCJ
yb2xlIjoiY2xpZW50ZSIsInV1aWQiOiIxIiwiZXhwIjoxNTExMzkzODk5fQ.v6s8moEB
CSMx_yaEuyTQ-tzoihT_S-f6g7Z3p8CmXbc'
    },
    data={
        "UserID": 0,
        "ProductID": alea,
```

```
            "ColorID": None,
            "SizesID": None,
        })

    def cart(l):
        l.client.request(method="GET",
        url="/esales/cart",
        headers={
            "Authorization":                                    'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiRGVib3JhaEhhbGwiLCJ
yb2xlIjoiY2xpZW50ZSIsInV1aWQiOiIxIiwiZXhwIjoxNTExMzkzODk5fQ.v6s8moEB
CSMx_yaEuyTQ-tzoihT_S-f6g7Z3p8CmXbc'
        })


    class UserBehavior(TaskSet):
        tasks = {index: 2, login: 1, cart: 4, listaproductos: 3, search:
5, detalle:6, checkout: 7, addcart: 8}

        def on_start(self):
            login(self)

    class WebsiteUser(HttpLocust):
        host = "http://35.194.27.142"
        task_set = UserBehavior
        min_wait = 5000
        max_wait = 9000
```

Anexo 5. Configuración de Locust en Kubernetes para la ejecución de las pruebas de carga.

Locust es una herramienta para realizar pruebas de carga a servicios web, está escrito en Python, la misma que puede ser fácilmente utilizada en Kubernetes de forma similar como montamos nuestros microservicios en nuestro clúster.

Para lo mismo necesitamos crear una imagen Docker, como se muestra, incluyendo Locust en ella.

Dockerfile

```
# Empezamos con una imagen base de Python 2.7.13
FROM python:2.7.13

MAINTAINER Beau Lyddon <beau.lyddon@realkinetic.com>

# Agregamos la carpeta con las tareas que ejecutará locust
RUN mkdir locust-tasks
ADD locust-tasks /locust-tasks
WORKDIR /locust-tasks

# Instalamos las dependencias que requiere Locust utilizando pip
RUN pip install -r /locust-tasks/requerimientos.txt

# Hacemos ejecutable el script bash .sh
RUN chmod 755 ejecutar.sh

# Exponemos los siguientes puertos para locust
EXPOSE 5557 5558 8089

# Ejecutamos Locust usando LOCUS_OPTS como variable de entorno
ENTRYPOINT ["./ejecutar.sh"]

```

El archivo Dockerfile, para crear la imagen Docker, se basará en Python 2.7.13, se le incluirá los archivos:

- locustfile.py
- requerimientos.txt
- ejecutar.sh

Todas las anteriores dentro de la carpeta *locust-tasks*.

Locust ejecutará las tareas asignadas en el archivo ***locustfile.py*** para realizar las pruebas de carga a nuestros microservicios.

Dockerfile ● locustfile.py ●

```python
from locust import HttpLocust, TaskSet
from random import randint

def login(l):
    l.client.post("/login",
    {
        "username": "donializ",
        "password": "password"
    })

def index(l):
    l.client.get("/")

def listaproductos(l):
    l.client.get("/products")

def search(l):
    l.client.get("/products/search?c=20&p=1&q=system")

def detalle(l):
    alea = randint(1, 50)
    l.client.get("/products/details/" + str(alea))
```

En ese archivo, *locustfile.py*, se programan las tareas de pruebas de carga, utilizando el lenguaje de programación Python según la documentación de Locust.

Los requerimientos de Locust son:

Flask, gevent, greenlet, itsdangerous, Jinja2, locustio, MarkupSafe, msgpack-python, pyzmq, requests, Werkzeug, urllib3 >= 1.15.1, certifi >= 14.05.14, y six.

Dichos requerimientos hacen que sea un poco pesada (tamaño que ocupa).

De manera similar como creamos la imagen Docker para cada microservicio, ejecutamos el comando ***docker build***:

docker build -t gcr.io/doctorado-182201/locust-tasks

```
$ docker images | grep locust
gcr.io/doctorado-182201/locust-tasks   latest
      7140a2f05734        2 days ago          714MB
```

La imagen creada pesará aproximadamente 714MB, a comparación de los microservicios, éste es muy grande, y tomará tiempo subirla al Google Container Repository para poder utilizarlo en Kubernetes.

gcloud docker -- push gcr.io/doctorado-182201/locust-tasks

Según las veces (versiones) que volvamos a subir modificaciones a nuestras tareas de pruebas de carga, algunas capas de requerimientos existentes de la imagen Docker creada serán reutilizadas, y sólo se subirán las que contengan los cambios, así ahorraremos tiempo al subir los cambios hechos a Locust.

A continuación, creamos archivos de configuración para crear Controladores de Replicación para Kubernetes, tanto para un servicio maestro de Locust y sus correspondientes esclavos.

La instancia maestra (master) unirá a las demás instancias de Locust, centralizando cada uno de ellos para mayor efectividad al momento de realizar las peticiones (pruebas de carga) a través de varias instancias (replicas) de Locust.

locust-master-controller.yaml

```yaml
kind: ReplicationController
apiVersion: v1
metadata:
  name: locust-master
  labels:
    name: locust
    role: master
spec:
  replicas: 1
  selector:
    name: locust
    role: master
  template:
    metadata:
      name: locust
      labels:
        name: locust
        role: master
    spec:
      containers:
        - name: locust
          image: gcr.io/doctorado-182201/locust-tasks:latest
          env:
            - name: LOCUST_MODE
              value: master
            - name: TARGET_HOST
              value: http://104.197.150.201
          ports:
            - name: loc-master-web
              containerPort: 8089
              protocol: TCP
            - name: loc-master-p1
              containerPort: 5557
              protocol: TCP
```

En la captura de código anterior podemos ver que se está utilizando la imagen de Locust que creamos anteriormente.

También se especifica la IP externa que Google Cloud Platform nos asignó al servicio, así Locust podrá encontrar los ***endpoints*** (APIs) a las que iremos enviando peticiones para la prueba de carga.

locust-master-controller.yaml ● locust-worker-controller.yaml ●

```yaml
kind: ReplicationController
apiVersion: v1
metadata:
  name: locust-worker
  labels:
    name: locust
    role: worker
spec:
  replicas: 10
  selector:
    name: locust
    role: worker
  template:
    metadata:
      labels:
        name: locust
        role: worker
    spec:
      containers:
        - name: locust
          image: gcr.io/doctorado-182201/locust-tasks:latest
          env:
            - name: LOCUST_MODE
              value: worker
            - name: LOCUST_MASTER
              value: locust-master
            - name: TARGET_HOST
              value: http://104.197.150.201
```

De manera similar se configura el controlador de replicaciones para los esclavos.

locust-master-service.yaml ● locust-master-controller.yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: locust-master
  labels:
    name: locust
    role: master
spec:
  ports:
    - port: 8089
      targetPort: loc-master-web
      protocol: TCP
      name: loc-master-web
    - port: 5557
      targetPort: loc-master-p1
      protocol: TCP
      name: loc-master-p1
    - port: 5558
      targetPort: loc-master-p2
      protocol: TCP
      name: loc-master-p2
    - port: 5559 ···
    - port: 5560 ···
    - port: 5561 ···
    - port: 5562 ···
    - port: 5563
      targetPort: loc-master-p7
      protocol: TCP
      name: loc-master-p7
  selector:
    name: locust
    role: master
  type: LoadBalancer

```

Finalmente, como se muestra en la captura de código anterior, configuramos un servicio de tipo ***LoadBalancer*** para que el servicio en la nube nos asigne otra IP externa destinado para Locust, haciendo accesible cada esclavo a través de diferentes puertos. Es importante mencionar que el puerto 8089, es el asignado al Locust maestro (master).

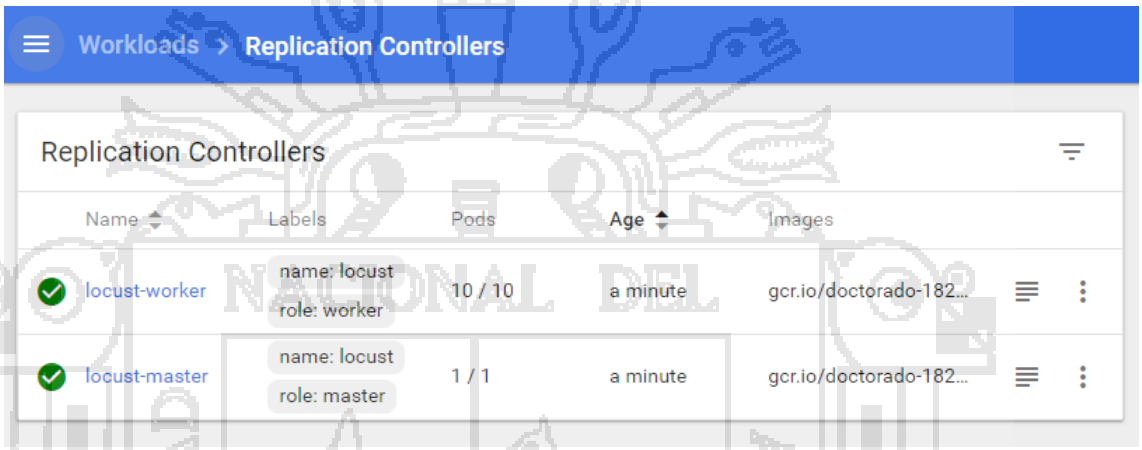
Ejecutamos los comandos correspondientes para subir cada una de las configuraciones a Kubernetes siguiendo el orden, primero el master, luego el esclavo (worker), finalmente el servicio que habilitará una IP externa para acceder a la interfaz visual de Locust a través de un navegador web.

kubectl create –f <master.yaml>

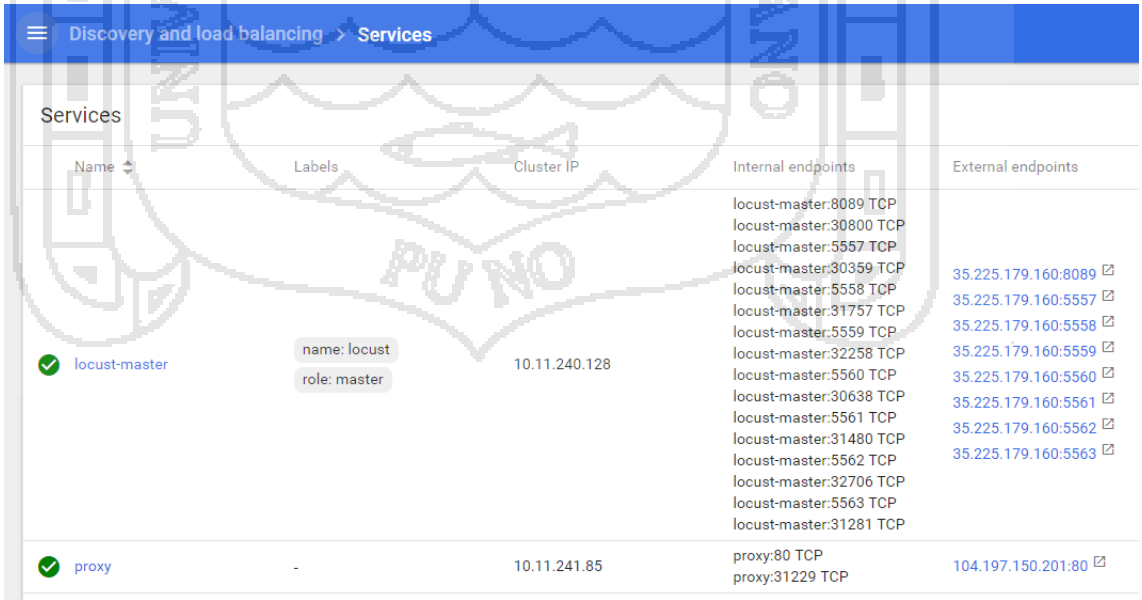
kubectl create –f <worker.yaml>

kubectl create –f <master-servicio.yaml>

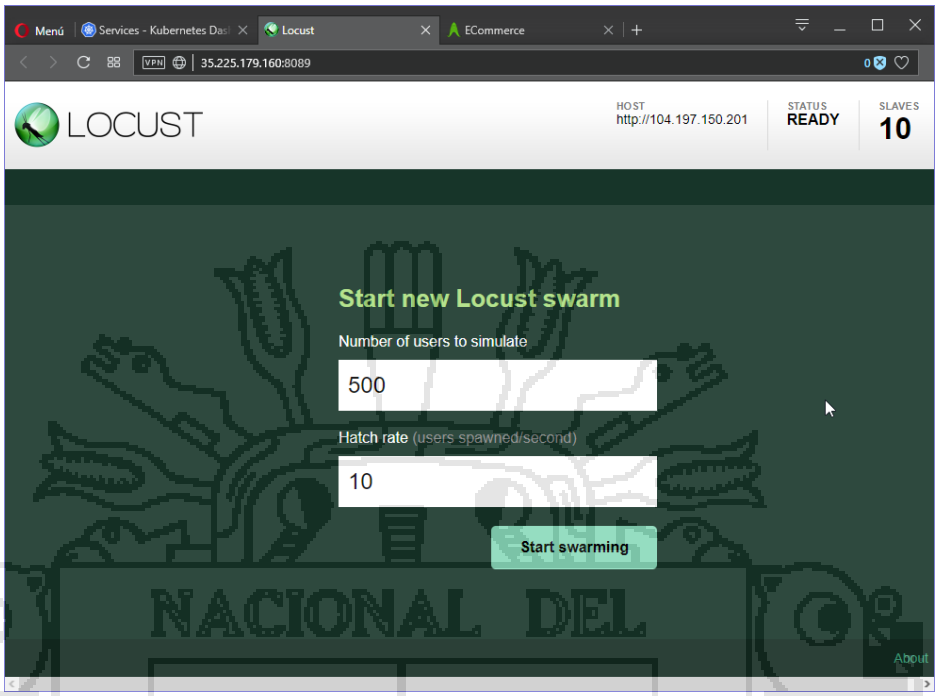
Cambiando cada nombre de archive entre símbolos "< >" con el nombre



Una vez ejecutado los comandos anteriores, veremos en la sección ***Workloads->Replication Controllers*** tanto al Locust maestro y los esclavos, ambos utilizando la misma imagen Docker de Locust que subimos con anterioridad.



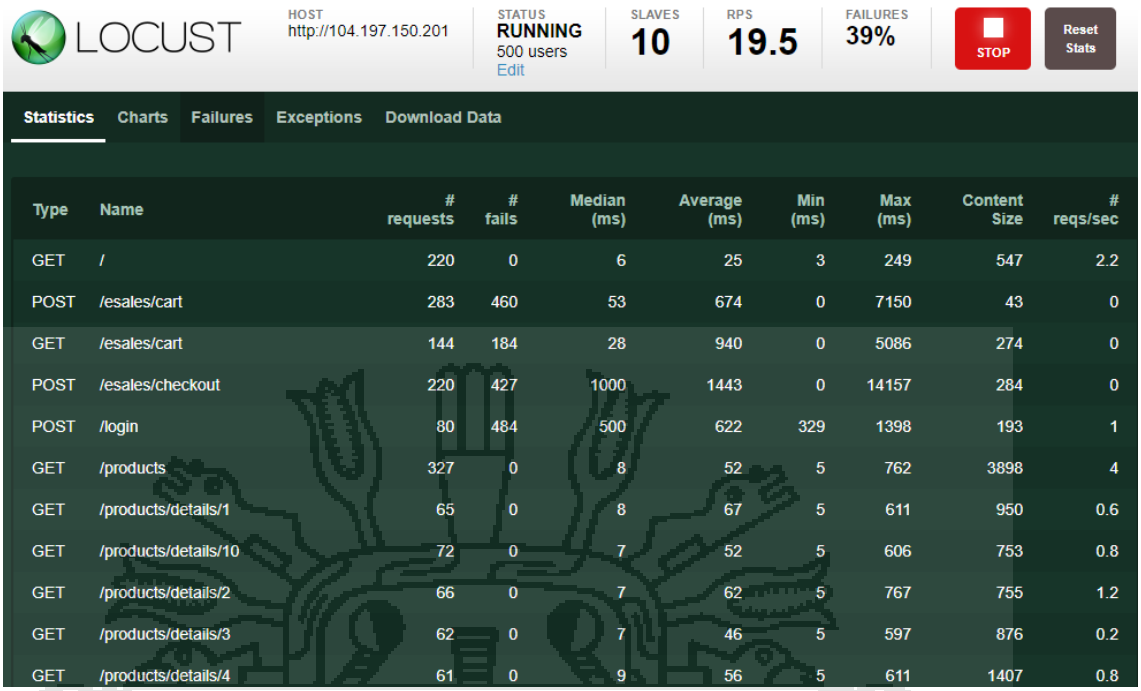
Además en la sección ***Discovery and load balancing->Services*** podremos apreciar que ***locust-master*** tiene asignada otra IP externa.



Al dirigirnos a esa nueva IP, la que tiene el puerto 8089, veremos que Locust se muestra visualmente en el navegador web, la misma que cuyo Host destino es la IP del Proxy (nuestro API Gateway).

Ingresamos la cantidad de usuarios a simular, con un radio de eclosión de 10 personas por segundo.

Al presionar el botón ***Start Swarming*** la prueba de carga se lanzará, utilizando 10 esclavos (instancias) haciendo peticiones todas al mismo tiempo a nuestra aplicación web.



Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size	# reqs/sec
GET	/	220	0	6	25	3	249	547	2.2
POST	/esales/cart	283	460	53	674	0	7150	43	0
GET	/esales/cart	144	184	28	940	0	5086	274	0
POST	/esales/checkout	220	427	1000	1443	0	14157	284	0
POST	/login	80	484	500	622	329	1398	193	1
GET	/products	327	0	8	52	5	762	3898	4
GET	/products/details/1	65	0	8	67	5	611	950	0.6
GET	/products/details/10	72	0	7	52	5	606	753	0.8
GET	/products/details/2	66	0	7	62	5	767	755	1.2
GET	/products/details/3	62	0	7	46	5	597	876	0.2
GET	/products/details/4	61	0	9	56	5	611	1407	0.8

Anexo 6. Datos obtenidos de las pruebas de carga.

Method	Name	# requests	# failures	Median response time	Average response time	Min response time	Max esponse ime	Average Content Size	Requests/s
GET	/	157	0	220	237	205	600	547	1.3
GET	/esales/cart	281	0	270	1329	208	6241	309	2.32
POST	/esales/cart	498	0	270	1075	215	7249	43	4.12
POST	/esales/checkout	399	0	470	1443	219	9358	283	3.3
POST	/login	498	0	270	1075	215	7249	43	4.12
GET	/products	265	0	230	257	215	962	7806	2.19
GET	/products/details/1	8	0	210	229	210	284	1020	12.3
GET	/products/details/10	11	0	210	222	209	297	1078	18.02
GET	/products/details/11	9	0	220	237	211	290	1403	12.3
GET	/products/details/12	11	0	230	257	211	373	1345	18.02
GET	/products/details/13	10	0	220	234	213	298	1400	14.2
GET	/products/details/14	9	0	220	269	211	410	1133	12.3
GET	/products/details/15	16	0	220	228	210	285	1149	3
GET	/products/details/16	6	0	220	227	212	287	1533	8
GET	/products/details/17	12	0	220	239	211	310	1468	20
GET	/products/details/18	9	0	220	246	210	387	1351	12.3
GET	/products/details/19	9	0	220	214	211	219	1139	12.3
GET	/products/details/2	10	0	220	235	210	297	1154	14.2
GET	/products/details/20	7	0	220	222	214	251	1153	9
GET	/products/details/21	14	0	220	237	210	330	1278	2
GET	/products/details/22	9	0	220	231	210	298	882	12.3
GET	/products/details/23	14	0	210	228	209	320	879	2
GET	/products/details/24	11	0	210	228	208	289	622	18.02

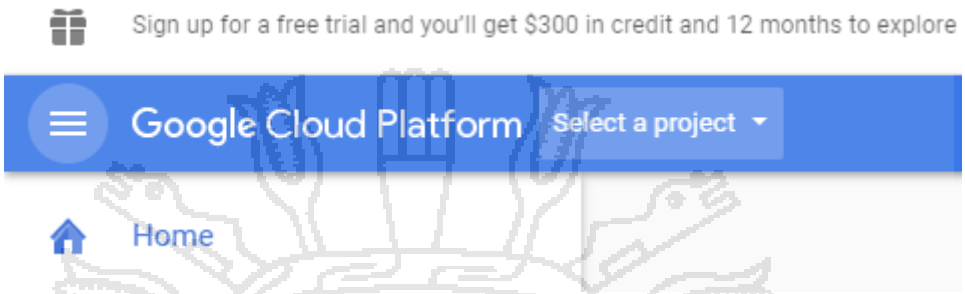
GET	/products/details/25	8	0	220	255	210	341	1218	12.3
GET	/products/details/26	9	0	220	229	209	306	1417	12.3
GET	/products/details/27	15	0	220	242	206	371	889	2
GET	/products/details/28	14	0	220	227	211	323	1278	2
GET	/products/details/29	8	0	220	238	212	313	1073	12.3
GET	/products/details/3	10	0	220	246	213	308	1214	14.2
GET	/products/details/30	10	0	210	234	207	297	1024	14.2
GET	/products/details/31	7	0	220	233	211	327	1226	9
GET	/products/details/32	16	0	220	230	209	305	1401	3
GET	/products/details/33	6	0	260	285	223	415	1404	8
GET	/products/details/34	7	0	220	229	214	307	1132	9
GET	/products/details/35	12	0	220	253	213	398	1154	20
GET	/products/details/36	11	0	220	242	212	403	1146	18.02
GET	/products/details/37	9	0	270	298	217	507	895	12.3
GET	/products/details/38	9	0	230	251	210	413	956	12.3
GET	/products/details/39	21	0	220	239	208	402	750	207
GET	/products/details/4	15	0	220	238	209	342	892	2
GET	/products/details/40	6	0	220	234	215	264	1269	8
GET	/products/details/41	10	0	220	219	209	265	1213	14.2
GET	/products/details/42	7	0	250	255	209	338	1393	9
GET	/products/details/43	14	0	230	258	212	400	1545	2
GET	/products/details/44	15	0	220	242	209	403	881	2
GET	/products/details/45	15	0	220	235	209	327	749	2
GET	/products/details/46	9	0	210	219	209	257	824	12.3
GET	/products/details/47	12	0	210	228	210	310	1089	20
GET	/products/details/48	10	0	210	231	211	335	1415	14.2
GET	/products/details/49	7	0	220	222	207	250	889	9
GET	/products/details/5	6	0	220	247	210	389	1396	8

GET	/products/details/50	11	0	250	263	215	368	891	18.02
GET	/products/details/6	12	0	210	235	209	436	892	20
GET	/products/details/7	9	0	220	233	209	334	886	12.3
GET	/products/details/8	6	0	220	290	214	406	1159	8
GET	/products/details/9	9	0	260	261	212	330	1097	12.3
GET	/products/search?c=20&p=1&q=system	465	0	230	260	215	1440	7724	3.84
None	**Total**	**3083**	**0**	**1309 0**	**1770 0**	**1203 5**	**4981 4**	**7339 9**	**762.6 9**

Anexo 7. Configuración de la infraestructura Kubernetes en Google Cloud

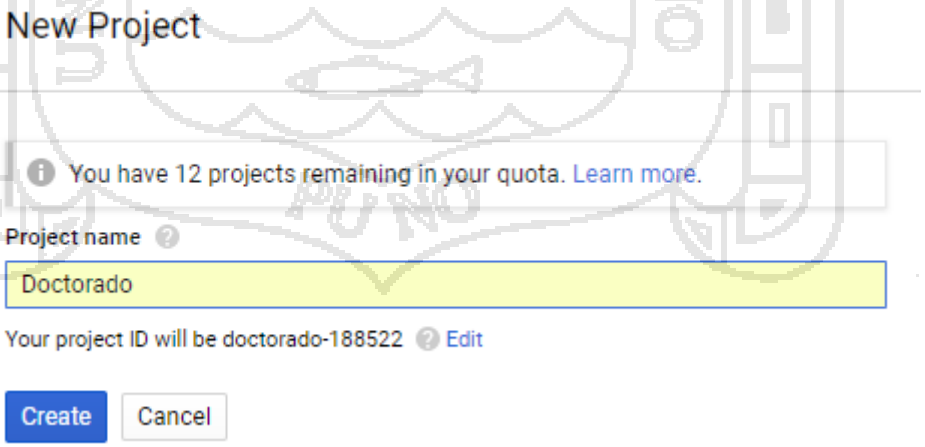
Dirigiéndonos a la página oficial de **Google Cloud Platform**[1], accedemos a través de nuestra cuenta de Google, sea éste Gmail u otro asociado al servicio.

Creamos el proyecto presionando el botón ***Select a project***.



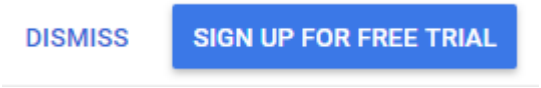
Nos mostrará un cuadro de diálogo, ahí recién podremos crear nuestro proyecto presionando el botón con el símbolo +.

Seguidamente llenamos los formularios de creación con los datos correspondientes según veamos por conveniente.

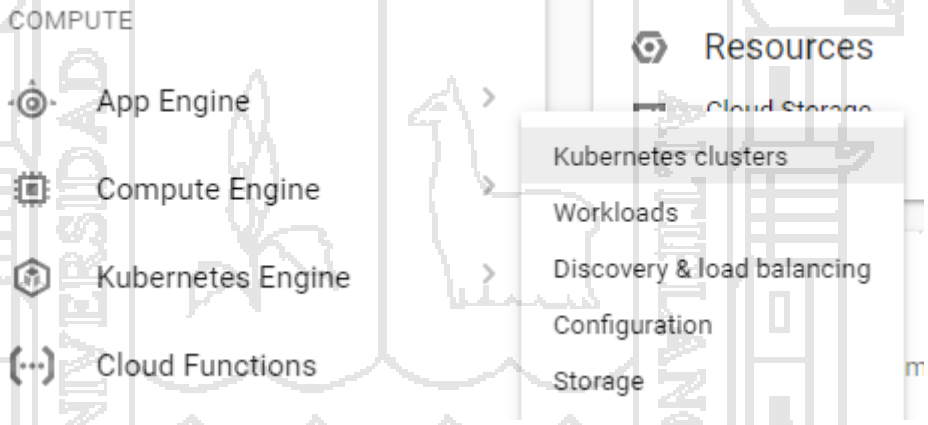


[1] https://console.cloud.google.com/home/dashboard

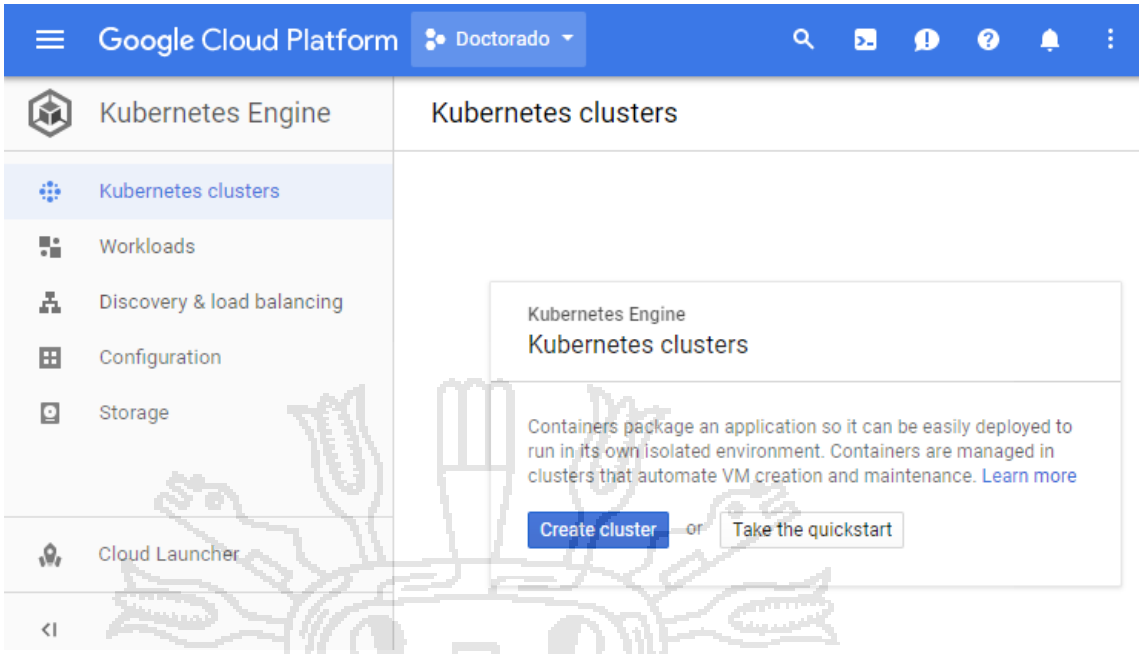
NOTA: Google Cloud Platform es de pago, pero también ofrecen periodos de prueba gratuito de 12 meses, para que podamos explorar sus características, lo cual lo podemos activar presionando el botón ***Sign up for free trial*** (Registrarnos para una prueba de tiempo limitado).



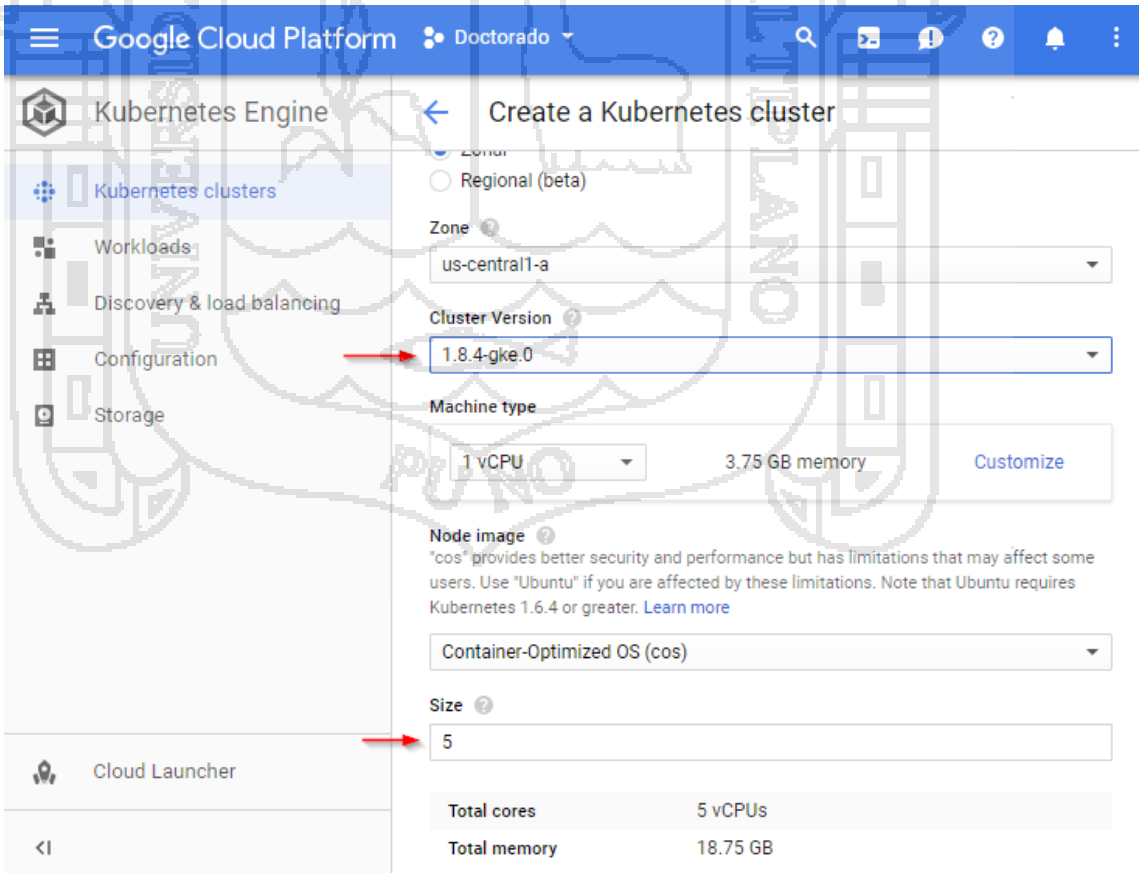
Terminado el proceso de activación del tiempo de prueba, habiendo ingresado nuestros datos personales y medios de pago electrónico, o si ya cuenta con una suscripción pagada, procedemos a levantar un nuevo clúster a través de Kubernetes, y para poder hacer uso de Kubernetes en la plataforma sobre la nube de Google, en la sección ***Compute*** elegimos ***Kubernetes Engine —> Kubernetes clusters***.



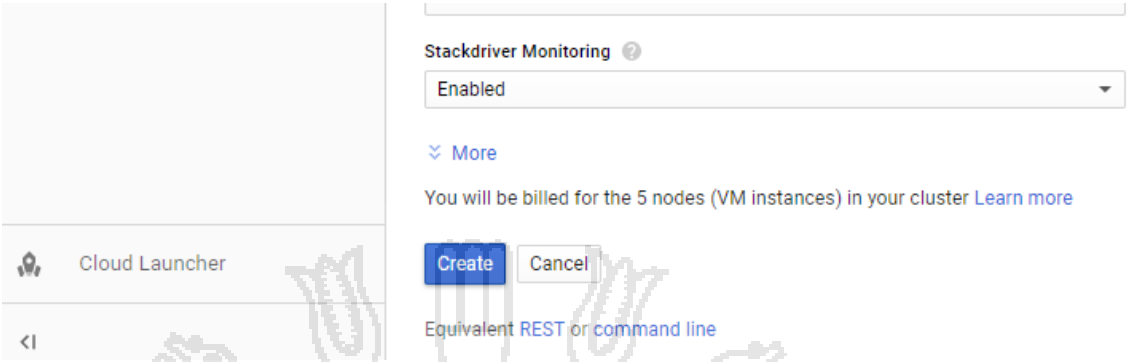
Ahí podremos levantar un *clúster* nuevo presionando el botón ***Create cluster***.



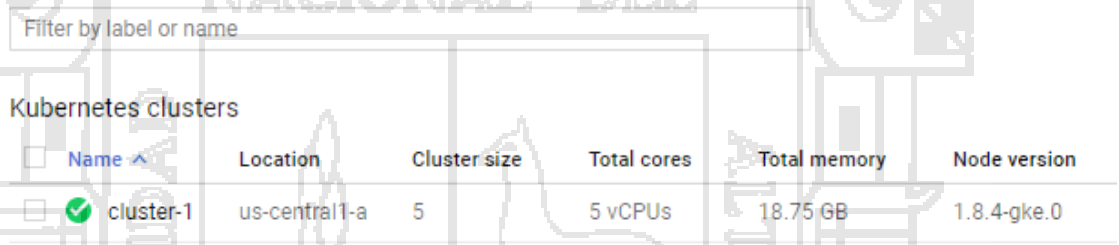
Se completa los datos según corresponda, teniendo en cuenta que es recomendable elegir la última versión del clúster (***Cluster version***) y un tamaño (***Size***) máximo de 5 nodos, debido a que el clúster gratuito para el periodo de prueba es no mayor o igual a 5 nodos, pasado éste límite se requiere de una cuenta con suscripción pagada.



Una vez satisfecho con la configuración realizada para el nuevo clúster, sólo faltaría presionar el botón ***Create***.



El tiempo de creación del clúster y levantado sus nodos para que ya podamos utilizar variará según el tamaño que hayamos elegido, y una vez terminada veremos algo similar a la siguiente imagen.



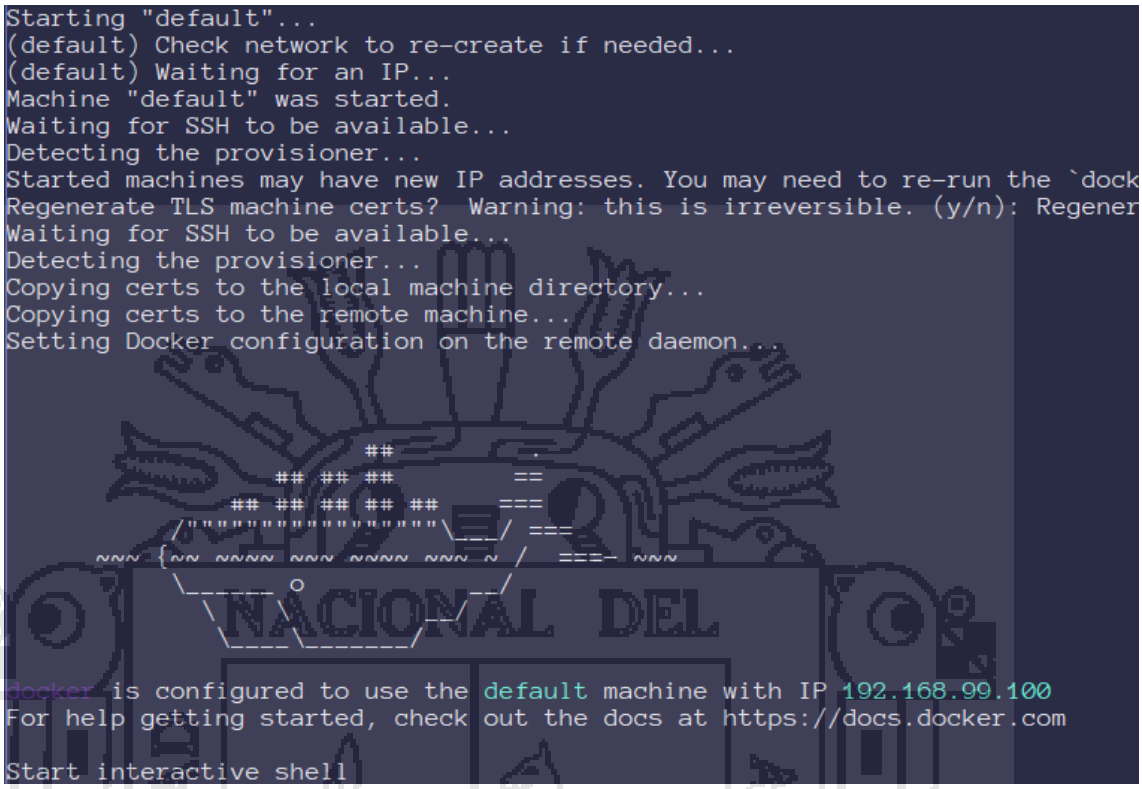
Con ello, ya tendremos listo el clúster donde podremos desplegar nuestro modelo de microservicios, para lo cual requerimos previamente haber instalado las herramientas que ofrece Google Cloud Platform para gestionar el clúster desde nuestra máquina local a través de internet.

La herramienta necesaria es **Google Cloud SDK**[2], la cual la debemos descargar e instalar.

Por otra parte, también requerimos tener instalado **Docker Toolbox**[3] junto con **VirtualBox**[4].

[2] https://cloud.google.com/sdk/
[3] https://docs.docker.com/toolbox/toolbox_install_windows/
[4] https://www.virtualbox.org/wiki/Downloads

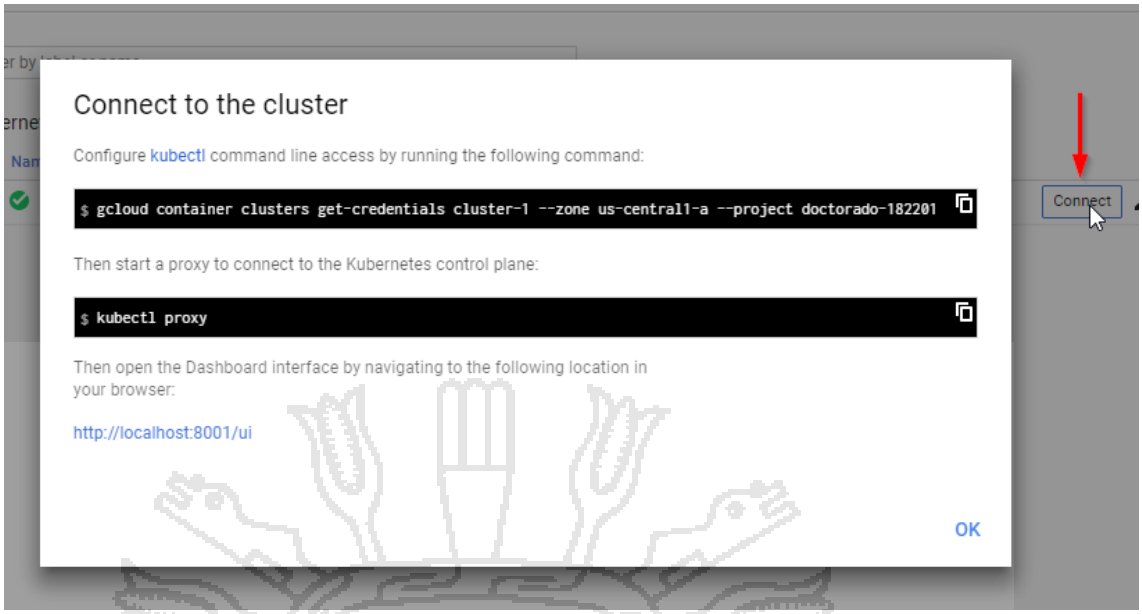
Para subir los Microservicios a la nube se empleó Docker Toolbox Terminal, que cuenta con comandos *bash* compatibles con Google Cloud SDK.



Inicializamos Google Cloud SDK con el comando **gcloud init**.

Esto procederá a conectarnos a nuestra cuenta a través de autenticación OAuth2, la misma que nos permitirá elegir el proyecto asociado a nuestro clúster, al terminar todos esos pasos de autenticación, sólo nos restaría acceder al panel de control a través de un proxy y así además poder gestionar nuestro clúster de forma visual a través del navegador web.

Para lo cual presionamos el botón ***Connect*** situado al costado del clúster que acabamos de crear anteriormente.



Nos desplegará detalles para conectarnos, tan sólo copiamos esos comandos para ejecutarlos en nuestro terminal Shell y las ejecutamos.

gcloud container clusters get-credentials cluster-1 --zone us-central1-a --project doctorado-182201

Esta línea de comandos asocia correctamente nuestro clúster actual con la configuración recientemente realizada a nuestra instalación de Google Cloud SDK.

Si todo va correctamente, el resultado devuelto sería similar a esto:

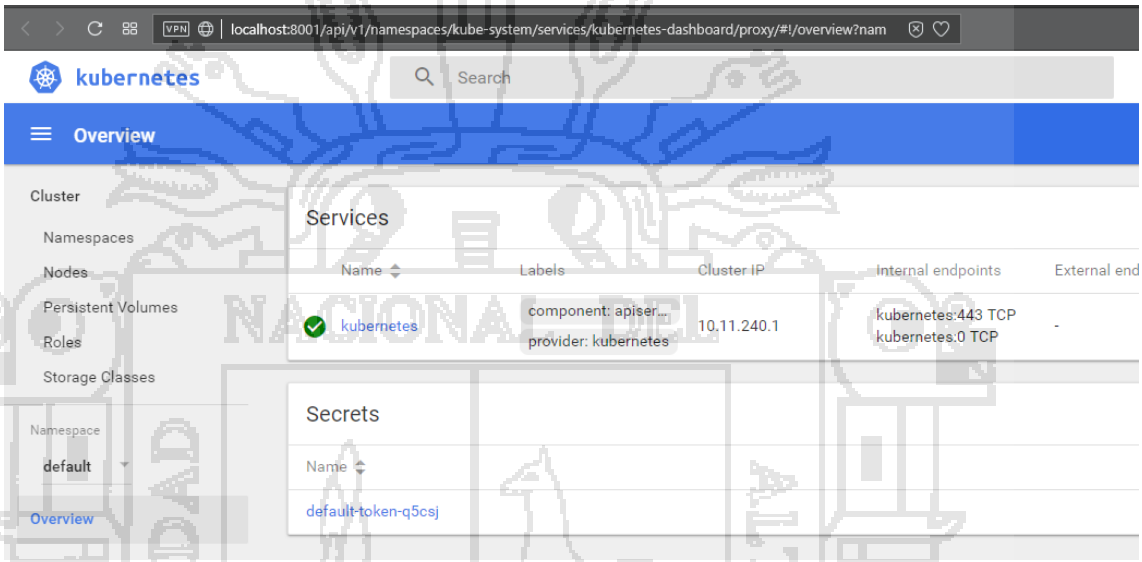
```
$ gcloud container clusters get-credentials cluster-1
Fetching cluster endpoint and auth data.
kubeconfig entry generated for cluster-1.
```

Ahora para ejecutar el proxy y así poder acceder de forma visual a través de un navegador web al panel de control (*Dashboard*) de nuestro clúster con Kubernetes, ejecutamos el terminal de Google Cloud SDK para no interferir con nuestro terminal actual.

Escribimos el comando ***kubectl proxy*** el que finalmente redirigirá el acceso a nuestro clúster a través de una conexión segura.

```
\AppData\Local\Google\Cloud SDK>kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Como se puede ver en la imagen anterior, el nuestro clúster ya puede ser accedido a través de la IP local 127.0.0.1 cuyo puerto asignado es el 8001.



Finalmente, se tiene correctamente configurada la conexión al clúster con Kubernetes, al cual ya podemos gestionarla a gusto.

Anexo 8. Despliegue de microservicios.

Para ello crearemos un documento de configuración con extensión ***.yaml*** donde configuraremos nuestro despliegue de microservicio en nuestro clúster.

Dockerfile ● {..} gusers.yaml ● {..} Configuración de usuario

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: usuarios
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: usarios
        track: stable
    spec:
      containers:
        - name: usuarios
          image: "gcr.io/doctorado-182201/usuarios"
          ports:
            - name: http
              containerPort: 82
            - name: health
              containerPort: 81
          resources:
            limits:
              cpu: 0.2
              memory: "20Mi"
          livenessProbe:
            httpGet:
              path: /healthz
              port: 81
              scheme: HTTP
            initialDelaySeconds: 5
            periodSeconds: 15
            timeoutSeconds: 5
          readinessProbe:
            httpGet:
              path: /readiness
              port: 81
              scheme: HTTP
            initialDelaySeconds: 5
            timeoutSeconds: 1
```

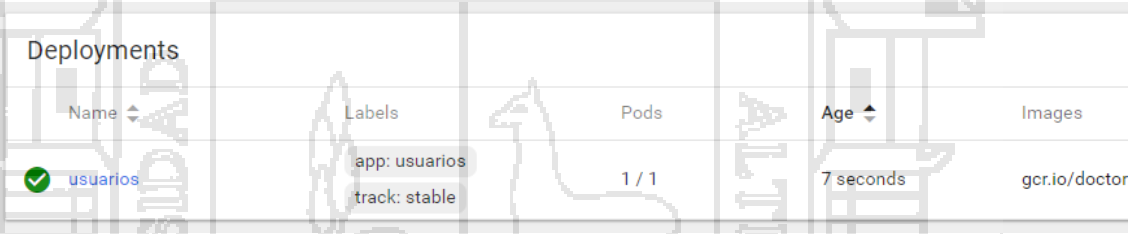
Entre las diferentes variables a configurar, las más importantes son el número de réplicas del microservicio, los puertos que deban coincidir con las utilizadas por

defecto de nuestro ejecutable embebido dentro de la imagen Docker que anteriormente realizamos. Además de la cantidad de memoria asignada, ciclos de reloj del CPU.

Una parte importante es la sección ***livenessProbe*** y ***readinessProbe***, que servirán para que ordenemos a Kubernetes el estar verificando la disponibilidad de nuestro microservicio, tanto su estado como su respuesta. La misma que al detectar problemas, éste repondrá otra copia del microservicio cuando éste no responda, lo más rápido posible.

Una vez configurado a gusto, ya podemos desplegar el microservicio en nuestro clúster utilizando el siguiente comando:

kubectl create -f gusuarios.yaml



Después de ello podemos verlo listado en la sección ***Deployments*** en el panel de control de Kubernetes. Se habrá de notar el nombre (*Name*), app:usuarios, y la imagen utilizada son las indicadas en el archivo de configuración que editamos anteriormente.

Estos despliegues además crearán *pods* y conjuntos de réplicas, según lo hayamos indicado en el documento .yaml que configuramos antes. Al ver el ***log*** de cada réplica, podremos percatarnos que éste corre perfectamente, y Kubernetes está verificando el estado del mismo.

```
Logs from usuarios  in usuarios-694c95967b-8m5zn
2017/12/10 22:24:17 Iniciando el servicio de Usuarios...
2017/12/10 22:24:17 Servicio del estado del servidor en %s 0.0.0.0:81
2017/12/10 22:24:17 Servicio HTTP escuchando en %s 0.0.0.0:82

(/mnt/c/Go/projects/src/github.com/doctorado/usuarios/main.go:34)
[2017-12-10 22:24:17]  [0.43ms]  INSERT INTO "users"
("created_at","updated_at","deleted_at","username","password","email","nombres","apellidos","sex
us","remember_token","avatar") VALUES ('2017-12-10 22:24:17','2017-12-10
22:24:17',NULL,'donializ','$2a$10$mL2OgqeL8AgnIljx4Hm4yOQGhDWLkgfeiQf3afAH4Mu9NGBx3x9XC','donial
001-01-01 00:00:00','','on','','')
[1 rows affected or returned ]

(/mnt/c/Go/projects/src/github.com/doctorado/usuarios/main.go:34)
[2017-12-10 22:24:17]  [0.13ms]  SELECT "locked" FROM "users"  WHERE (id = '1')
[1 rows affected or returned ]
10.8.3.1:48608 - - [Sun, 10 Dec 2017 22:24:23 UTC] "GET /healthz HTTP/1.1" kube-probe/1.8+
10.8.3.1:48610 - - [Sun, 10 Dec 2017 22:24:24 UTC] "GET /readiness HTTP/1.1" kube-probe/1.8+
10.8.3.1:48638 - - [Sun, 10 Dec 2017 22:24:34 UTC] "GET /readiness HTTP/1.1" kube-probe/1.8+
10.8.3.1:48648 - - [Sun, 10 Dec 2017 22:24:38 UTC] "GET /healthz HTTP/1.1" kube-probe/1.8+
```

Nótese, que este microservicio se ejecuta en el puerto 82, mientras que utiliza el puerto 81 para informar el estado del mismo a Kubernetes.

Sin embargo, al tener el microservicio ejecutándose, aún no podemos utilizarlo en conjunción con los demás microservicios, por tanto es necesario utilizar el servicio de descubrimiento de Kubernetes a través de K8DNS, asignándole un nombre con el que se pueda rastrear fácilmente, para más tarde a través de un ***API GATEWAY*** podamos redireccionarlo a una única dirección IP.

Para crear el servicio, también recurrimos a hacer utilizando un archivo de configuración como el siguiente:

Dockerfile ● {..} gusers.yaml ● {..} usersvc.yaml ●

```yaml
kind: Service
apiVersion: v1
metadata:
  name: "usuarios"
spec:
  selector:
    app: "usuarios"
  ports:
    - protocol: "TCP"
      port: 80
      targetPort: 82
```

Ahí especificamos el nombre cómo será identificado por el DNS de Kubernetes, además de redirigir su acceso a través del puerto 80 en lugar del que utilizamos por defecto, que era el 82. E indicamos el nombre de la ***app*** correspondiente al despliegue que hicimos anteriormente.

Finalmente, con el siguiente comando enviamos el contenido del fichero como parámetro para que en nuestro clúster con Kubernetes cree el servicio.

kubectl create -f usuariosvc.yaml

Una vez ejecutado tal comando, en la sección ***Discovery and load balancing->Services*** de Kubernetes podremos ver listado los diferentes microservicios, con IPs que variarán cada vez que éstas se recreen. Por lo tanto, el nombre (*Name*) nos será de utilidad para ubicarlo sin necesidad de recordar IPs.

El microservicio *frontend* o aplicación web, que incluye la parte visual para que la aplicación web muestre al usuario final, está escrito utilizando VueJS como *framework* en lenguaje JavaScript, Sass y otros (imágenes, fuentes tipográficas).

El resultado es una aplicación web de estilo SPA (Single Page Application) o aplicación de una página única. La misma que realiza peticiones a través de AJAX a los ***endpoints*** (APIs) que expone cada microservicio, de tal manera interactúa con cada microservicio que será accesible a través del API Gateway (ver siguiente anexo).

La aplicación será empacada también en una imagen Docker, incluyendo además un servidor web compatible con la revisión del estado que utilizamos en Kubernetes.

Para lo cual utilizamos el mismo lenguaje Golang, para crear el servicio de archivos en lugar de utilizar otros programas servidores web como Apache, Nginx, Caddy, etc.

Compilamos nuestro propio servidor web para que se acceda al *frontend* tal cual como lo hicimos para los microservicios anteriores.

main.go Dockerfile

```
1  FROM alpine:3.1
2  MAINTAINER Somebody
3  ADD frontend /usr/bin/frontend
4  RUN chmod 0755 /usr/bin/frontend
5  COPY public /public
6  ENTRYPOINT ["frontend"]
7
```

El Dockerfile para el *frontend* incluirá tanto el ejecutable denominado de la misma forma (frontend), e incluirá la carpeta de archivos necesarios de la aplicación web (HTML, Javascript, hojas de estilo, fuentes tipográficas, imágenes, entre otras).

Generamos la imagen Docker y la subimos a Google Container Registry de forma similar como subimos los microservicios.

En cuanto al despliegue del *frontend* en Kubernetes, lo hacemos de similar manera como hicimos con los microservicios, ya que en realidad es un ejecutable con acceso a los archivos de la aplicación web accesibles en la misma imagen Docker.

Anexo 9. Intercomunicación de los microservicios.

Una parte importante a mencionar, es la comunicación entre microservicios, al momento de obtener datos necesarios de otro microservicio, el microservicio debe utilizar el nombre de dominio interno asignado al microservicio objetivo en Kubernetes, es decir el nombre de servicio asignado al microservicio.

La cual pasamos, para más comodidad, como parámetro al ejecutable del microservicio, debido a que escribir ese nombre en el código fuente y compilarlo de forma rígida, es poco conveniente.

Esto no es relevante para el despliegue del microservicio, pero sí permite configurar nuestra imagen Docker.

main.go × Dockerfile sales ● Dockerfile fro

```
1  FROM alpine:3.1
2  MAINTAINER Somebody
3  ADD ventas /usr/bin/ventas
4  ENTRYPOINT ["ventas","--domain","productos"]
5
```

En la imagen anterior, se puede ver el Dockerfile del microservicio **ventas**, la misma que incluye a su ejecutable llamado de la misma forma, pero se le pasa como parámetro el nombre del microservicio **productos**, el cual internamente apunta al dominio **http://productos** que gracias al servicio DNS de Kubernetes, este microservicio ubicará fácilmente al otro microservicio y mediante RESTful obtendrá información que ésta requiera para sus propósitos, haciéndose de esta manera la comunicación entre microservicios.

Es muy importante, por tanto, asignarle un nombre de servicio a nuestros microservicios, lo cual facilita mucho su descubrimiento por otros microservicios, en comparación a utilizar IPs desconocidas, que además varían según Kubernetes levante otra réplica de reemplazo a cada despliegue de microservicio que termina asignando otra IP, mientras que al utilizar un nombre de dominio (servicio), será más fácil de ubicar.

Anexo 10. Configuración del API GATEWAY basado en NGINX

Utilizaremos NGINX como *reverse proxy* la misma que servirá de API Gateway para unir los diferentes microservicios a través de una misma IP y desde luego asignándoles ***endpoints*** adecuados según cómo hayamos indicado en cada uno de ellos a la hora de programarlos.

Para tal cometido configuramos el archivo ***proxy.conf*** para utilizarlo con Nginx**.**

Dockerfile ● {..} gusers.yaml ● {..} usersvc.yaml ● proxy.conf

```
upstream frontend {
    server frontend.default.svc.cluster.local;
}

upstream productos {
    server productos.default.svc.cluster.local;
}

upstream usuarios {
    server usuarios.default.svc.cluster.local;
}

upstream ventas {
    server ventas.default.svc.cluster.local;
}

server {
    listen 80;

    location /productos {
        proxy_pass http://productos;
    }

    location /login {
        proxy_pass http://usuarios;
    }
    location /usuarios {
        proxy_pass http://usuarios;
    }

    location /esales {
        proxy_pass http://ventas;
    }

    location / {
        proxy_pass http://frontend;
    }
}
```

Como se habrá de notar en el documento, se especifica el nombre acorde a los asignados a los servicios creados para cada despliegue de microservicio. Redirigimos cada uno a su correspondiente *endpoint* con la variable *proxy_pass* recurriendo al DNS con el nombre de dominio de cada microservicio. Todo ello accesible a través del puerto 80.

Este archivo de configuración debe ser utilizado en otro despliegue y también ser accesible mediante otro servicio, de forma similar como hicimos para cada microservicio.

Dockerfile ● {..} proxy.yaml ✕ {..} gusers.yaml ● {..} users

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: proxy
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: proxy
        track: stable
    spec:
      containers:
        - name: nginx
          image: "nginx:1.9.14"
          lifecycle:
            preStop:
              exec:
                command: ["/usr/sbin/nginx","-s","quit"]
          volumeMounts:
            - name: "nginx-proxy-conf"
              mountPath: "/etc/nginx/conf.d"
      volumes:
        - name: "nginx-proxy-conf"
          configMap:
            name: "nginx-proxy-conf"
            items:
              - key: "proxy.conf"
                path: "proxy.conf"
```

Con la diferencia que éste utilizará una imagen Docker del repositorio oficial de Docker (para obtener el programa servidor web y proxy llamado Nginx) y el archivo ***proxy.conf*** que subiremos al servicio de ***ConfigMap*** de Kubernetes con el nombre ***nginx-proxy-conf*** como se puede ver en el documento ***proxy.yaml*** (ver imagen anterior).

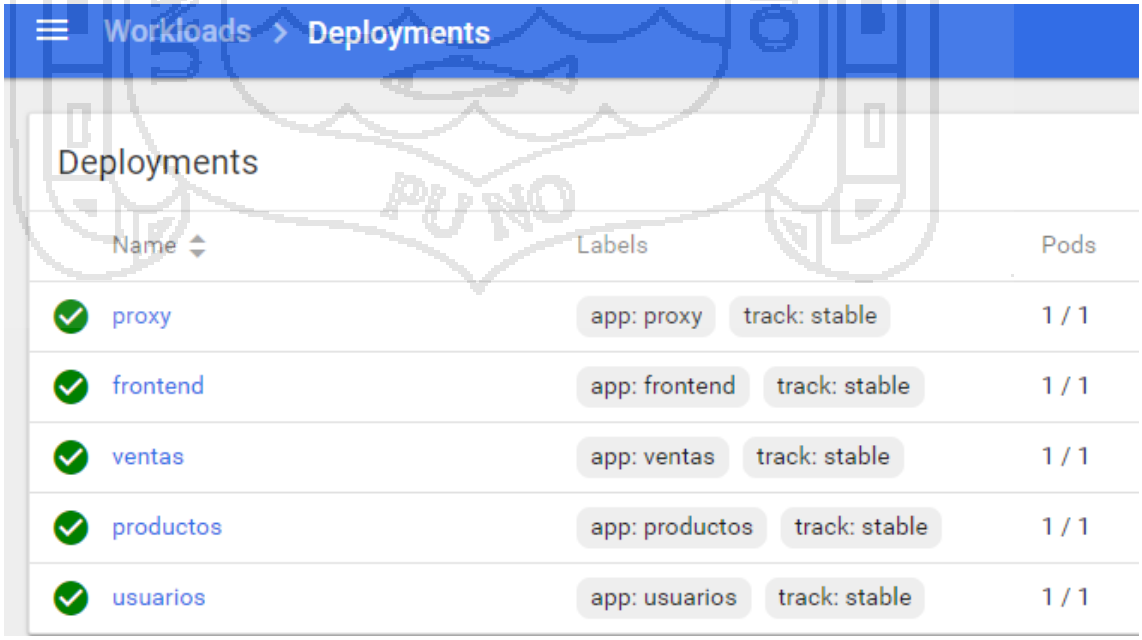
Sin embargo, es necesario subir primero el archivo de configuración ***proxy.conf*** antes de desplegar nuestro ApiGateway, así evitar que falle al no encontrarlo en el ***ConfigMap*** (mapa de configuración) de Kubernetes.

Con el siguiente comando enviamos el archivo de configuración que utilizaremos para ejecutar Nginx (***proxy.conf***):

kubectl create configmap nginx-proxy-conf --from-file=proxy.conf

Podremos ver que el archivo *proxy.conf* ya estará listado con el nombre que especificamos en la sección ***Config and Storage->ConfigMaps*** de Kubernetes.

Ahora ya podemos crear el despliegue de forma similar como desplegamos los microservicios: **kubectl create -f proxy.yaml**



Como se ve en la captura anterior, nuestro ApiGateway (proxy) está listado junto a los demás microservicios que desplegamos más antes. Finalmente, para tener acceso a nuestra aplicación web a través de un navegador web, necesitamos que el servicio de Google Cloud Platform nos asigne una IP estática pública a este proxy, ya que este une a todos los demás microservicios, incluyendo la interfaz de usuario visual (frontend), y las demás como *endpoins* (APIs RESTFul), que interactúan entre sí como si estuvieran en una misma máquina (hosting). La configuración del proxy como servicio de acceso público la realizamos como sigue:

proxysvc.yaml

```
1   kind: Service
2   apiVersion: v1
3   metadata:
4     name: "proxy"
5   spec:
6     selector:
7       app: "proxy"
8     ports:
9       - protocol: "TCP"
10        port: 80
11        targetPort: 80
12    type: LoadBalancer
```

Como se notará, es tan similar como los servicios creados para cada microservicio, con la diferencia que éste es de tipo ***LoadBalancer***, la cual Kubernetes utilizará para informar al servicio de clúster de Google Cloud Platform para que finalmente éste nos asigne una IP pública. El comando a utilizar es similar como para la creación de servicios que antes usamos: **kubectl create -f proxysvc.yaml**

Discovery and load balancing > Services

Services

Name	Labels	Cluster IP	Internal endpoints	External endpoints
proxy	-	10.11.241.85	proxy:80 TCP proxy:31229 TCP	104.197.150.201:80
frontend	-	10.11.248.73	frontend:80 TCP frontend:0 TCP	-
ventas	-	10.11.246.54	ventas:80 TCP ventas:0 TCP	-
productos	-	10.11.246.235	productos:80 TCP productos:0 TCP	-
usuarios	-	10.11.246.19	usuarios:80 TCP usuarios:0 TCP	-
kubernetes	component: apiserver provider: kubernetes	10.11.240.1	kubernetes:443 TCP kubernetes:0 TCP	-

Una vez ejecutado tal comando, el servicio tomará un tiempo (no tan largo) para que el servicio de Google Cloud Platform nos asigne una IP pública. Como se puede apreciar en la imagen anterior, el servicio cuenta con una IP (External endpoints) pública, a diferencia de los demás.

Al dirigirnos a esa IP a través de un navegador web, ya podremos ver en funcionamiento nuestra aplicación web utilizando el modelo composición microservicios, donde el carrito de compra funciona a través de la composición de tres microservicios que son usuarios, productos y ventas.