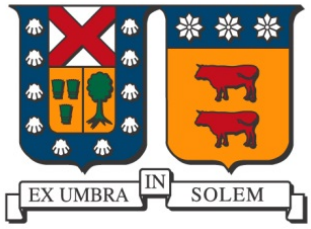
2022-01

PROPUESTA DE ARQUITECTURA DE SOFTWARE E INFRAESTRUCTURA PARA DESPLEGAR APLICACIÓN MÓVIL DE RECOMENDACIÓN BASADA EN SERENDIPIA PARA EMPRENDEDO

OLGUÍN BARAZARTE, JUAN CARLOS ESTEBAN

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“PROPUESTA DE ARQUITECTURA DE SOFTWARE E INFRAESTRUCTURA PARA DESPLEGAR APLICACIÓN MÓVIL DE RECOMENDACIÓN BASADA EN SERENDIPIA PARA EMPRENDEDORAS”

JUAN CARLOS ESTEBAN OLGUÍN BARAZARTE

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Víctor Codocedo
Profesores Correferentes: Pablo Baeza y Claudia López

ENERO - 2022

AGRADECIMIENTOS

Agradezco profundamente el apoyo y consuelo de mis amigos, en particular de Gabriela y Geordy quienes son mis amigos más cercanos de la universidad, a los chicos del labcomp por la buena onda y por siempre estar con la disposición de compartir experiencias que puedan ayudar a uno en el trabajo, disposición que he tratado de replicar por mi parte hacia quienes estimo por lo que me apasiona de esta disciplina. A mi familia; a mi madre Sandra y a mi padre de mi mismo nombre por su apoyo incondicional y a mi hermana Nathalie que fue y es un pilar fundamental de mi estado emocional.

También a mis incondicionales amigos que he hecho en el transcurso de mi vida con quienes hasta el día de hoy hablamos a pesar de todo el tiempo y circunstancias ocurridas… ustedes saben quienes son, si los nombrara uno a uno quizás no terminaría nunca.

Y por último a aquellos con quienes he compartido de una u otra forma pero que por diversas razones nuestros caminos divergieron, créanme o no, parte de ustedes crece en mi hasta el día de hoy.

Si hay algo que he aprendido en mi paso por la universidad, es que los amigos valen oro.

RESUMEN

Resumen— Diseñar una arquitectura de software e infraestructura para una aplicación móvil que asegure cumplir con los requisitos no funcionales en especial para cuando se debe de satisfacer a una considerable cantidad de potenciales clientes.
En el presente trabajo se desarrollará una propuesta de arquitectura de software e infraestructura para solucionar este problema en el despliegue de la lógica de negocios de una aplicación móvil que es basado en serendipia para emprendedoras.
Se analizarán distintos paradigmas de arquitectura de *software* e infraestructura para analizar cómo utilizar estos conceptos en desarrollar y desplegar apropiadamente la aplicación, donde en función del análisis del contexto del proyecto que se tiene actualmente se propone utilizar una arquitectura de microservicios dentro de lo posible, desplegando los servicios en contenedores y exponiéndolos a través de una *API gateway*.

Palabras Clave— ***microservicios; despliegue; operaciones; contenedores; arquitectura de software***

ABSTRACT

Abstract— Designing the architecture and infrastructure of a mobile application that assures with some of the usual non-functional requirements it's always a challenge to pursue in the design of those systems, specially if it is important to satisfy a potentially huge quantity of users or clients.
In the present study, a software and infrastructure architecture approach to prevent these non-functional requirements to not be satisfied is proposed for the deployment of the business logic of a mobile app which is based on serendipity for women entrepreneurs.
Some software and infrastructure architecture paradigms will be analyzed to check how to start developing and deploying appropriately this system, where in function of the analisys on the project context, the proposed architecture of the system will be based on microservices when possible, deploying the services in containers and exposing them through an API gateway.

Keywords— microservices; deployment; operations; containers; software architecture

GLOSARIO

ANID: Agencia Nacional de Investigación y Desarrollo.
DI: Departamento de Informática.
uSCI: Unidad de Servicios de Computación e Internet
UTFSM: Universidad Técnica Federico Santa María.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Las prácticas de desarrollo y despliegue de servicios van cambiando en el tiempo mientras se van descubriendo nuevas formas de implementar software en función de las capacidades computacionales que han crecido constantemente a través de las últimas décadas.

En el presente trabajo de memoria, se propondrá una arquitectura de software e infraestructura iniciales que idealmente puedan presentar una solución, materializar y hacer realidad la aplicación móvil Aliadas, que es una aplicación de generación de contactos entre compradores y vendedores.

Esto se realiza con el fin de otorgar de la mejor forma posible con los recursos otorgados de capacidad de computo para el proyecto con los ejes de seguridad, rendimiento, mantenibilidad y escalabilidad aparte de desarrollar los requisitos funcionales de la lógica de negocios de la aplicación con buenas prácticas de desarrollo y despliegue de aplicaciones, dando a lugar para ello una implementación de arquitectura de software basada principalmente en microservicios desplegado con una arquitectura de infraestructura basado principalmente en despliegue de contenedores para ejecutar los servicios que permiten el funcionamiento de la aplicación Aliadas.

Para ello se tuvo que pasar por un proceso creativo para revisar como dividir la lógica de negocios de la aplicación en distintas aplicaciones independientemente desplegables y escalables, es decir que se puedan ejecutar con concurrencia para poder por ejemplo priorizar la cantidad de instancias de un servicio según se utilice más, o menos en comparación con los demás servicios.

Los resultados fueron favorables considerando que se esperaban 50 usuarios registrados en la aplicación el día de lanzamiento.

En el presente trabajo de memoria, se realizará inicialmente una breve definición del problema, para posteriormente revisar los arquetipos de arquitectura de software e infraestructura posibles junto a un análisis de estos según la bibliografía en el marco conceptual para posteriormente proponer de las arquitecturas de software e infraestructura estudiadas las posibles soluciones para el problema del desarrollo y despliegue de la aplicación Aliadas junto a un análisis de varias tecnologías que facilitarán la materialización de la aplicación. Luego se realizará una evaluación de la arquitectura de software usando una metodología para el desarrollo de aplicaciones modernas, y pruebas de estrés, para posteriormente comparar el trabajo realizado con otros exponentes de la industria del desarrollo de aplicaciones web y móviles en el estado del arte para finalmente realizar una retrospectiva y análisis sobre las decisiones tomadas, recomendaciones y el trabajo futuro y pendiente para el proyecto.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

1.1. Preámbulo

Se presenta la situación del desarrollo de una aplicación móvil de un proyecto que consiste en una red social de generación de contactos entre emprendedoras con funcionalidades de *marketplace*, es decir, de interacción entre compradores, tiendas, vendedores y sus productos, que es un proyecto financiado por la ANID (Agencia Nacional de Investigación y Desarrollo) y gestionado por la profesora Claudia López, el profesor Víctor Codocedo y Paulina Santander, además de un equipo de ingenieros, desarrolladores y practicantes donde el autor trabaja en el despliegue de la aplicación, servicios y desarrollo *back-end*. Se estableció que la presente aplicación tenga una demanda inicial de alrededor de 50 usuarios en horario pico en el lanzamiento de la aplicación y se entrevistaron 11 emprendedoras [1] por lo que definir una arquitectura de software e infraestructura adecuada es menester para aportar en su éxito.

1.2. Caracterización del problema

Para materializar el proyecto tanto en lo que es afín a este trabajo y a lo que no lo es que es el diseño de interfaz, recolección de requisitos y entre otros, hay distintos equipos que trabajaron en el proyecto Aliadas, están los equipos de diseño, datos y *back-end*, jefe de proyecto y los profesores que dirigen el proyecto. En el equipo de datos que es en el que pertenece el autor del presente trabajo hay 6 personas; 2 practicantes, 3 memoristas y 1 ingeniera titulada que se añadió durante el desarrollo de la aplicación.

La presente aplicación se espera que tenga una potencial alta demanda a futuro por lo que definir una arquitectura de software e infraestructura adecuada es menester para aportar en su éxito.

La lógica de negocios de la aplicación, que es la parte que se encarga de abstraer la solución de los problemas del mundo real, debe de estar definida bajo una especificación de arquitectura de software e infraestructura: la arquitectura de software considera principalmente el diseño del código para solucionar el problema de la aplicación Aliadas; por otro lado, la arquitectura de infraestructura especifica los equipos computacionales y conexiones de red y herramientas necesarias para lograr satisfacer las necesidades de los usuarios de la aplicación.

[1] El trabajo de las entrevistas se encuentra en la memoria para obtener el título de Ingeniera Civil Industrial de Camila Torres Muñoz

La necesidad de especificar bien estas dos arquitecturas (también “diseñar una arquitectura” o “definir una arquitectura”) surge de la complejidad asociada a una plataforma que soporta varias aplicaciones de software, de distinta naturaleza, con diferentes requerimientos, desarrollados por diferentes equipos, al tiempo que sirve a múltiples y diferentes usuarios con niveles de calidad de servicio conocidos, considerando además, políticas de seguridad que deben ser garantizadas tanto en el despliegue de la plataforma como en el desarrollo de cada uno de sus componentes.

Para ilustrar este punto, podemos contrastar las necesidades del desarrollo de una plataforma *ecommerce*, a aquellas que puede tener el desarrollo de una calculadora personal. En el último caso, no es necesaria la especificación de una arquitectura de software dada la simpleza de las operaciones que deben ser implementadas (por ejemplo, suma, resta, multiplicación y división de dos dígitos) las cuales pueden ser parte de una misma aplicación. Más aún, tal aplicación podría ser implementada fácilmente en un mismo código fuente en una arquitectura denominada como “monolítica” (que en este caso no requiere de una especificación). Al contrario, un *ecommerce* se compone de un conjunto de aplicaciones que interactúan entre ellas (aplicaciones de autenticación, aplicaciones de recomendación, aplicaciones de bases de datos, aplicaciones Web, etc). Una arquitectura de software, en este caso, nos permite especificar cómo estas aplicaciones deben interactuar, cuáles de ellas deben ser desarrolladas o cuáles de ellas pueden ser implementadas por software de terceros. En el caso de aquellas que deban ser desarrolladas, la arquitectura define en líneas generales las funciones que deben cumplir (requerimientos funcionales) y los estándares bajo los cuales deben cumplirlas (requerimientos no funcionales).

De la misma forma, el desarrollo de un software para una calculadora personal no requiere la especificación de una arquitectura de infraestructura, pues la lógica de la aplicación puede residir fácilmente en un sólo dispositivo de cómputo (e.g. un teléfono móvil, un computador o incluso un reloj de pulsera). Al contrario, la plataforma de *ecommerce* considera la ejecución simultánea y concurrente de las distintas aplicaciones mencionadas anteriormente en una familia de dispositivos (teléfonos móviles, computadores personales, servidores físicos, servidores virtuales, etc). La arquitectura de infraestructura debe definir entonces en qué tipo de dispositivo debe ejecutarse qué aplicación y en base a qué requerimientos. Por ejemplo, una aplicación de base de datos debe ejecutarse en un servidor físico o virtual con suficiente espacio de almacenamiento, mientras que una aplicación de recomendación más bien debe ejecutarse en un servidor físico o virtual con suficiente poder de cómputo.

De manera más formal, según **Martin Fowler** [Fowler, 2019] una arquitectura de software bien diseñada soporta la propia evolución del código a largo plazo y es fácil de modificar o iterar para agregar nuevas características, caso contrario a una arquitectura de software mal diseñada. Por otro lado una arquitectura de infraestructura bien diseñada permite ejecutar de forma ordenada, eficiente y con la menor latencia posible, un conjunto de uno o más servicios evitando caídas de el o los servicios desplegados, además de maximizar la facilidad en la mantenibilidad de los equipos computacionales.

El diseño de estas arquitecturas corresponde a un proceso de selección y adaptación de pa-

trones o paradigmas adecuados a la problemática a resolver, en este caso, los sistemas que compondrán la plataforma de Aliadas. Estos patrones son evaluados considerando sus respectivas ventajas y desventajas. Al mismo tiempo, se tienen que considerar las limitantes de recursos existentes para la ejecución del proyecto y adecuar el diseño de estas arquitecturas a estas limitantes.

La selección del patrón de arquitectura de software se realiza considerando los servicios y componentes que son parte del ecosistema del sistema de software a construir. A su vez, los servicios y componentes del sistema se definen en función de los requerimientos o requisitos funcionales de la solución. En el proyecto Aliadas, los requisitos funcionales, son recopilados por el equipo de diseño y luego son comentados y discutidos en reuniones con el equipo de datos y *back-end*, estos requisitos permiten definir las funcionalidades del sistema como un conjunto de entradas y salidas de datos que un sistema debe cumplir, una referencia a estos requisitos se puede encontrar en la sección de anexos, en estos requisitos es que se basa el diseño de servicios y selección de componentes sobre el cual se profundizará próximamente dentro del presente documento.

En estas reuniones se toman decisiones de cómo satisfacer estos requisitos, definiendo desde ellos un conjunto de servicios y componentes.

La selección de una arquitectura de software depende de las características de estas que se han de considerar, las cuales se mencionan en [M., 2015], la caracterización de cada arquitectura de software se compara a través de los ejes de agilidad para reaccionar a cambios de demanda de uso, facilidad de despliegue, propenso a pruebas de las funcionalidades del código, rendimiento, escalabilidad y facilidad de desarrollo. Notar que algunos de las características mencionadas calzan con algunos requerimientos no funcionales que se introducirán posteriormente, esto da a notar que la arquitectura de software a implementar influye directamente en el cumplimiento de los requisitos no funcionales. Ninguna arquitectura es la solución absoluta a todos los problemas de desarrollo de software, si por ejemplo tenemos el caso del desarrollo de una página informativa con texto plano, que se sabe que no será tan visitada, o que apunte a un nicho de público, tal vez se debería de apuntar más a una arquitectura de software que aporte a la facilidad de desarrollo y despliegue que a una que ofrezca más escalabilidad por ejemplo, contrario al caso de una página de *e-commerce* concurrida constantemente para comprar productos.

Como ya se mencionó, la elección de un patrón de arquitectura de software influye directamente con los requisitos no funcionales, y como se verá después, la de infraestructura también. La arquitectura de software definida comprende los servicios o componentes que se deben de desplegar, por lo que en función de esto se debe de determinar un patrón de despliegue de el o los servicios que componen la plataforma. La implicación de tener e implementar una arquitectura de software es evitar lo que se llama un anti-patrón llamado “gran bola de lodo” como se menciona en [M., 2015] que consiste en código mal organizado y con módulos con falta de roles claros, además de dificultad para ser editado sin crear fallas y como consecuencia una gran dificultad para comprender el código sin tener una comprensión a través del estudio profundo de este lo cual viene netamente de malas prácticas y falta de

organización.

Una vez definida la arquitectura de software, se consideran las limitantes bajo las que se encuentra el desarrollo y despliegue de la lógica de negocios de la aplicación, las cuales corresponden a *los requerimientos o requisitos no-funcionales* sobre las cuales profundizaremos en la siguiente sección.

En este caso, en particular, las limitantes son, las de tiempo en el que se debe de desarrollar la solución, las limitantes económicas del proyecto (considerando los recursos existentes otorgados por ANID) y la capacidad de cómputo que se tendrá para el despliegue de los servicios, además de la cantidad de personal y horas trabajadas en el equipo de datos y *back-end*.

La dificultad de tomar decisiones respecto a en que patrones de arquitectura basarse, radica en las ventajas y desventajas de cada uno lo cual se verá más adelante en el marco teórico, aparte, se debe de hacer un cruce con las limitantes de recursos computacionales que se tienen para esto.

Ahora, sobre la arquitectura de infraestructura, esta es necesaria, debido a que sin recursos computacionales, el o los servicios definidos en la arquitectura de software no podrán ser ejecutados, es necesario tener infraestructura suficiente y adecuada para poder ejecutar el *back-end* de la aplicación Aliadas, tal y como por ejemplo, un videojuego requiere de recursos computacionales para ser ejecutado.

Además el, recordando el objetivo principal de este es permitir el despliegue de él o los servicios, en este caso de *back-end* utilizando los recursos computacionales y conexiones de red necesarias como se mencionó anteriormente y utilizando si es oportuno, herramientas de despliegue que faciliten la disponibilidad de él o los servicios a desplegar.

Por otro lado es importante tener en cuenta sobre cuantos recursos se necesitan de computo, la documentación de las tecnologías y componentes tienden a mostrar claramente la cantidad de recursos que se necesitan para desplegar estos, entonces es menester tomar en consideración estos requisitos técnicos para levantar la arquitectura de infraestructura, sino, puede ser que o se invierta demasiado en recursos computacionales o que llegue a faltar para desplegar la arquitectura de software. Con esto es imprescindible añadir que una arquitectura de infraestructura es construida en función de los componentes y servicios que se deben de levantar para manifestar la arquitectura de software.

Es importante recalcar que para cumplir los requisitos no funcionales se deben de seguir prácticas sugeridas idealmente por la bibliografía, esto debido a que son metodologías, protocolos o políticas practicadas y confirmadas que ayudan a cumplir con los requisitos no funcionales, estas políticas y su aplicación se explicaran durante el transcurso del texto.

Todos estos factores influyen directamente en las decisiones a futuro dentro del proyecto Aliadas, en particular en la definición de arquitecturas para confeccionar la solución.

Los desafíos que se deben enfrentar en este trabajo son el diseño e implementación de ar-

quitecturas de software e infraestructura que se mantengan estables y mantenibles en el tiempo en función de las limitantes mencionadas, la definición de las tecnologías que se usarán para cumplir este cometido y sobre esto último, definir cuándo desarrollar una funcionalidad o utilizar una tecnología existente que permita al equipo ahorrar tiempo para facilitar el trabajo de desarrollo.

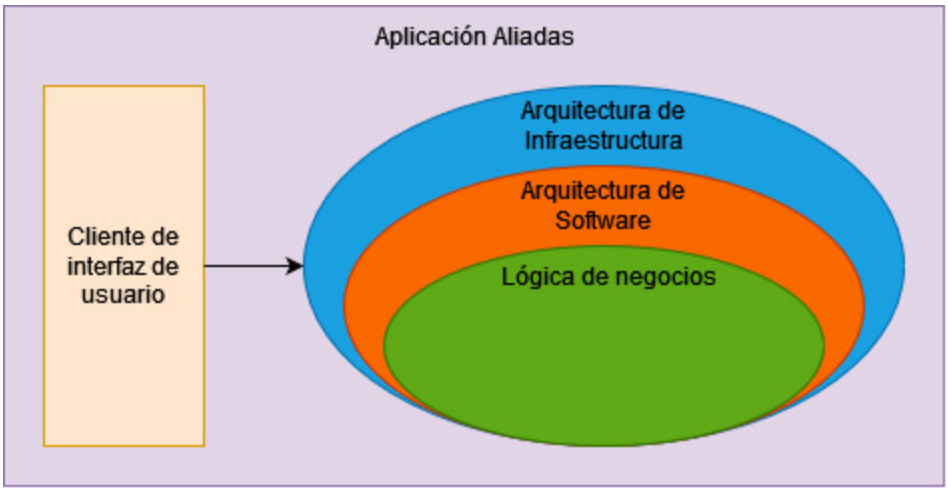


Figura 1: Las arquitecturas de software e infraestructura con sus respectivos módulos y componentes deben de ser definidas para el correcto funcionamiento de la aplicación Aliadas
Fuente: Elaboración propia

1.3. Caracterización de requisitos no funcionales

Para poder aterrizar y caracterizar los problemas a solucionar en el desarrollo y despliegue de la lógica de negocios de la aplicación, se observarán los requisitos no funcionales sugeridos por el Instituto Internacional del Análisis de Negocios (o en inglés *International Institute of Business Analysis*) [IIBA, 2015] para aplicaciones *e-commerce* y se adaptarán levemente para las necesidades de la aplicación Aliadas ante la falta de una caracterización de requisitos no funcionales.

Estos son:

- **Usabilidad**: El sitio (o aplicación encargada de la presentación) debe ser utilizable incluso y sobre todo por usuarios que no tienen conocimientos técnicos.
- **Seguridad**: Se deben hermetizar los roles de los usuarios de la aplicación apropiadamente para que así no se puedan vulnerar las acciones de estos, también se habla de hermetizar transacciones sin embargo **Aliadas no manejará transacciones monetarias al menos para cuando este proyecto esté desplegado en el desarrollo de este trabajo de memoria**.

- **Rendimiento**: La aplicación debe ser práctica y realizar las consultas pertinentes lo más rápido posible, para entregar una mejor experiencia de usuario.
- **Mantenibilidad**: La lógica de negocios debe ser lo más sencillo posible de mantener, las fallas son inevitables a largo plazo en estos tipos de sistema, por lo que se debe de abaratar costos en la mantención de la aplicación.
- **Escalabilidad**: Se deben considerar soluciones a prueba de incremento de usuarios a futuro, estas soluciones definirán el crecimiento de la aplicación en función del añadido de funcionalidades sin pasar a llevar el rendimiento de la aplicación.

En el presente trabajo, las soluciones desarrolladas y desplegadas se basarán en facilitar los últimos 4 requisitos no funcionales, osea, ”Seguridad, Rendimiento, Mantenibilidad y Escalabilidad”. Esto debido a la naturaleza de los servicios que se desarrollaran y desplegarán en este trabajo pues se trabajará netamente en la lógica de negocios y en el despliegue de este sin involucrarse con la interfaz de usuario que es parte de otro trabajo de memoria de un miembro del equipo de diseño [2].

1.4. Objetivo General

Proponer y desplegar una arquitectura de software e infraestructura con buenas prácticas de despliegue considerando requerimientos de seguridad, rendimiento, mantenibilidad y escalabilidad.

1.5. Objetivos Especificos

1. Proponer y desplegar una Arquitectura de Software considerando los requerimientos funcionales y no funcionales del proyecto Aliadas.
2. Proponer y desplegar una Arquitectura de Infraestructura considerando los requerimientos funcionales y no funcionales del proyecto Aliadas.
3. Definir políticas y mecanismos de ambas arquitecturas para cumplir con los requisitos no funcionales definidos para el proyecto vinculados a seguridad, rendimiento, mantenibilidad y escalabilidad.

[2] Si se desea saber más del diseño de la interfaz de este proyecto, consultar la memoria para optar al título de Ingeniero Civil Informático de Eduardo Reyes.

CAPÍTULO 2
MARCO CONCEPTUAL

En lo que sigue, se presenta un desarrollo del marco conceptual necesario para describir el desarrollo de este trabajo de memoria.

2.1. Arquitectura de software

La arquitectura de software es la definición de un conjunto de soluciones que permite optimizar atributos relacionados a una serie de decisiones, ejemplos de estos atributos son seguridad, rendimiento y mantenibilidad (como los requisitos no funcionales mencionados en la definición del problema).

Este conjunto de decisiones impactará directamente en la calidad de la aplicación y en el posterior éxito de este.

2.1.1. ¿Qué tipos de arquitectura de software existen?

En [M., 2015] se describen 5 tipos de arquitectura de software, todas con sus respectivas ventajas y desventajas las cuales se resumen a continuación.

- Arquitectura Monolítica (o de N capas):

  Un sistema con arquitectura monolítica se compone de capas (*layers*) aisladas (las más usuales son la capa de presentación, la capa de negocios, la capa de persistencia y la capa de base de datos) las cuales se unen para formar el sistema o aplicación en una pieza en un solo conjunto de código desplegable.

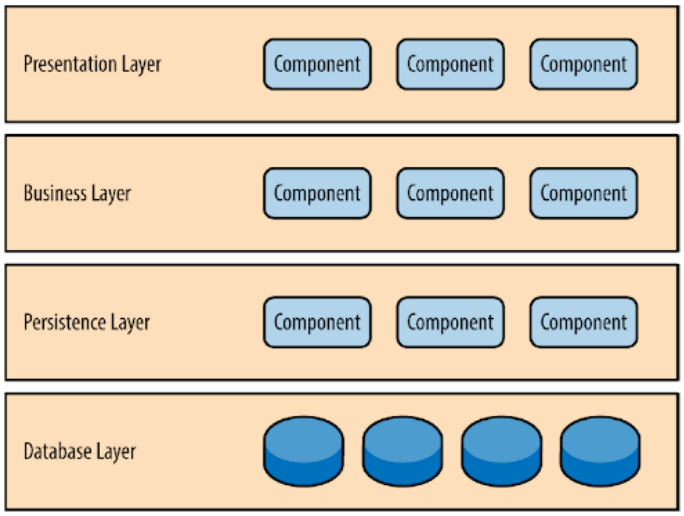


Figura 2: Ilustración de arquitectura monolítica
Fuente: [M., 2015]

A continuación se describirán las capas usuales de una arquitectura monolítica que se muestran en la figura 2, estas capas están usualmente divididas en componentes de código.

- Capa de presentación:
  Esta capa tiene que ver con la implementación de código asociado a la interacción entre el usuario y sus intenciones con el sistema, es por donde se ingresa datos para obtener información desde la capa de negocio de la aplicación.

  Ejemplo de componentes: Un tablero de información en la página de un banco, o una lista de contactos de un *chat*.

- Capa de Negocio:
  Es aquella que maneja la lógica del dominio de la solución de la implementación de la aplicación, siendo el dominio el conjunto de entidades abarcados y abstraídos en el sistema para dar una solución a este.

  Ejemplo de componentes: Una clase “Usuario”, el controlador que llama a los métodos para modificar las instancias de este y la definición de rutas para utilizar estos métodos desde la capa de presentación.

- Capa de persistencia:
  Es aquella parte que abstrae y maneja la persistencia de datos en el o las bases de datos necesarios para el correcto funcionamiento del sistema.

Ejemplo de componentes: Las ORM (*Object Relational Mapping*) que permiten abstraer las consultas de bases de datos en clases que heredan de los modelos basados en el dominio de la aplicación.

- Capa de base de datos:
  Es aquella que tiene la lógica necesarias para conectarse a la base de datos apropiadamente a través usualmente de *URLs*.

  Ejemplo de componentes: Paquetes o extensiones que permiten conectarse a una base de datos como por ejemplo *motor* que permite conectarse a una base de datos **MongoDB** usando el lenguaje **Python**

- Arquitectura conducida por eventos: Este tipo de arquitectura utiliza eventos para activar y comunicarse entre servicios desacoplados.

  La arquitectura conducida por eventos consiste en la producción por un servicio productor de eventos y consumo por un servicio que consume estos eventos asociados a por ejemplo, una acción del usuario dentro de una aplicación.

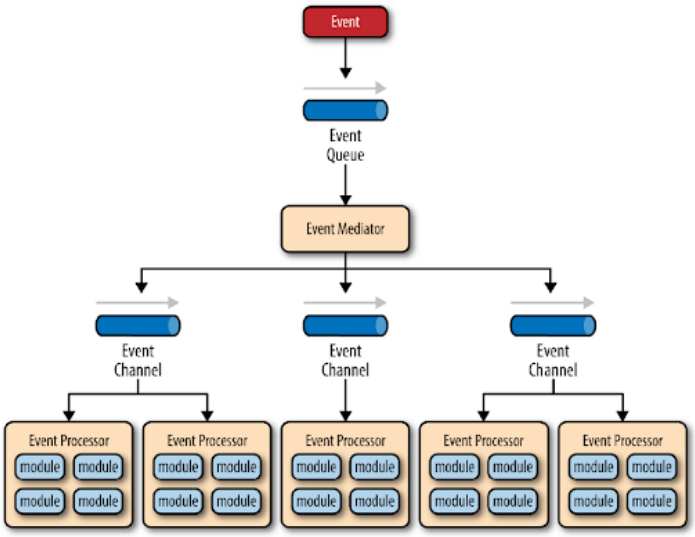


Figura 3: Ilustración de arquitectura conducida por eventos
Fuente: [M., 2015]

Como se muestra en la figura 3 se ingresa el evento a una cola que captura estos eventos, este evento puede ser por ejemplo una consulta hecha a través de una *API*. Una vez en la cola, este se manda a un mediador de eventos que según el tipo de evento, lo manda a un canal distinto para ser procesado posteriormente por alguna aplicación desplegada que esté escuchando a esa cola de eventos.

Uno de los usos de este patrón es para mantener consistencia en la persistencia de datos entre servicios, imaginemos por ejemplo que tenemos un servicio de usuarios con su propia base de datos en el que se intenta borrar un usuario, el evento entonces

sería el borrado del usuario, este evento es mandado a la cola de eventos para posteriormente ser mandado al canal de evento asociado al respectivo evento, una vez este evento llega al servicio que esta encargado de persistir alguna entidad asociada al usuario, como por ejemplo, el avatar. se procede a borrar los avatares asociados al usuario. Notar que esto asegura la consistencia de los datos ya que si por ejemplo si no hay alguna instancia del servicio de imágenes operativo, este evento es guardado en la cola hasta que alguna instancia de este servicio esté arriba. En el presente trabajo se utilizo en particular este caso de uso para la aplicación.

- Arquitectura de microkernel (o arquitectura de *plugins*):

  Este tipo de arquitectura consiste en la interacción entre un sistema central y un conjunto de módulos (o *plugins*). La lógica de negocios se divide en módulos independientes, mientras que el sistema central, siendo este último la aplicación con las funcionalidades principales y los módulos, extensiones de este para casos más específicos.

  Por ejemplo: el sistema central puede ser un *browser* y un módulo de este puede ser Adblock, un módulo para el bloqueo de publicidad.

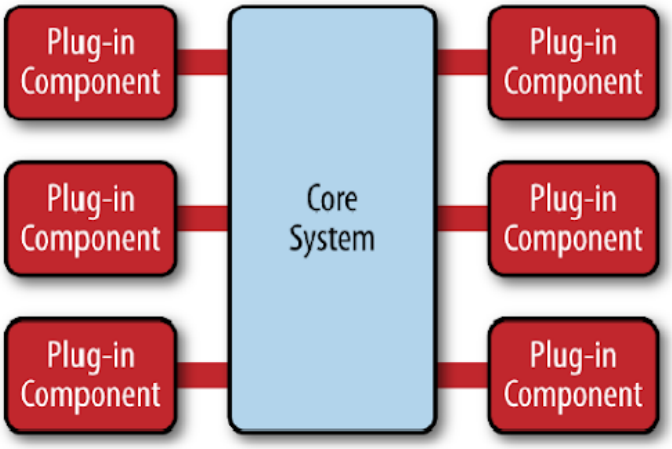


Figura 4: Ilustración de arquitectura de microkernel
Fuente: [M., 2015]

- Arquitectura de Microservicios:

  Este tipo de arquitectura consiste en la división de las funcionalidades del sistema que se despliegan independientemente entre sí en distintas aplicaciones desacopladas. Este tipo de arquitectura es el opuesto de una arquitectura monolítica donde se tienen todas las funcionalidades dentro de una aplicación.

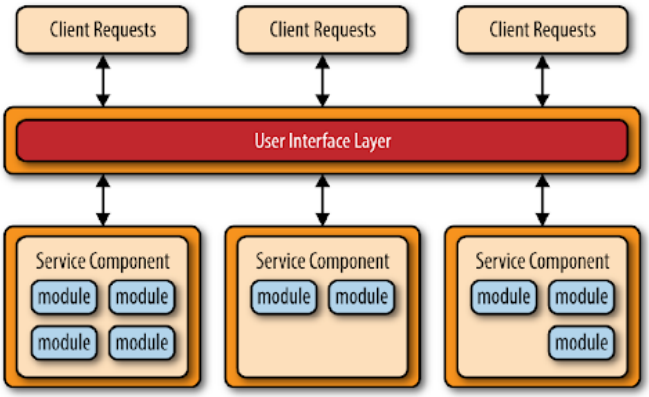


Figura 5: Ilustración de arquitectura basada en microservicios
Fuente: [M., 2015]

Como se aprecia en la figura 5, se reciben consultas de los clientes a través de una capa de interfaz de usuario donde a través de el se consultan a los distintos servicios que componen el sistema para cumplir con la función de la aplicación.

Si esta aplicación usara una arquitectura monolítica, todos los componentes de servicios y la capa de interfaz de usuario se juntarían en una sola aplicación.

De hecho la capa de interfaz de usuario sería el equivalente a la capa de presentación y los componentes de los servicios se encargarían de la capa de negocios, persistencia y base de datos.

En la bibliografía analizada [M., 2015] se realiza un cuadro comparativo entre 4 arquitecturas mostrado en la figura 6, la última es la arquitectura *cloud*, la cual se mencionará mas tarde, debido a su naturaleza de arquitectura de despliegue más que de arquitectura de software en el contexto de esta memoria.

Es importante destacar que las arquitecturas de software dentro de una aplicación no son excluyentes unas de las otras, es decir, un sistema puede adoptar más de un patrón de arquitectura.

Por ejemplo una arquitectura de microservicios puede perfectamente usar servicios que tengan características de una arquitectura de microkernel o consumir un servicio *cloud* sin dar complicaciones en su diseño.

Otro ejemplo es el que se realiza en el presente proyecto, donde se usa una arquitectura de microservicios y aparte se usa una arquitectura conducida por eventos para mantener la consistencia entre algunos componentes de servicios, esto se explicará en mayor detalle en la propuesta de solución.

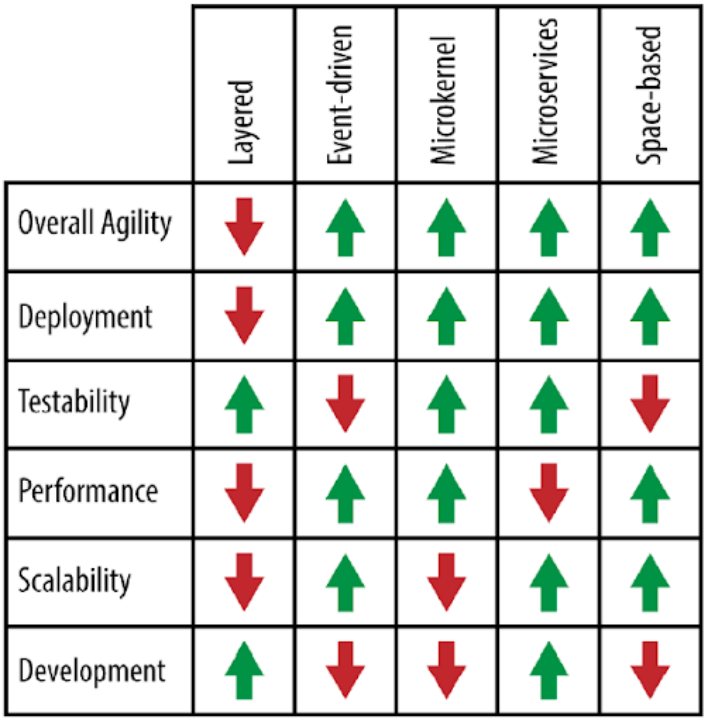


Figura 6: Resumen de análisis de arquitecturas de software
Fuente: [M., 2015]

2.1.2. Análisis de Escalabilidad

Sin embargo, hay que tener en cuenta un problema de la arquitectura monolítica y es que se puede escalar con facilidad en un solo eje, en particular en el eje X. Se puede escalar en el eje Z pero se necesita modificar código específicamente para esto, lo cual es mejor evitar a menos que el dominio de la aplicación dicte que vale la pena su implementación (por ejemplo si tuvieramos usuarios *VIP* y usuarios normales), sin embargo, es mejor desacoplar la lógica de la escalabilidad del software y dejar eso para la definición de la infraestructura de la aplicación. Por otro lado una arquitectura de microservicios presenta facilidad para escalar adecuadamente en los ejes X osea, desplegando nuevas instancias de los servicios e Y osea, dividiendo el dominio de la aplicación en servicios desacoplados entre si, esto sin la necesidad de escalar en el eje Z, lo cual es un proceso menos práctico de implementar, debido a que es ideal que la lógica de la infraestructura y del software queden desacoplados para a futuro facilitar el desarrollo y despliegue de los servicios.

Por lo tanto, en función a la cantidad de clientes que se espera que tenga la aplicación, una arquitectura *netamente* monolítica a largo plazo, no será compatible con los requisitos no funcionales ligados al proyecto debido a la dificultad que implica escalar en el eje Y la aplicación si se sigue este paradigma, lo cual nos deja en particular con la arquitecturas de microservicios en el ámbito de arquitecturas de software.

2.2. Arquitectura de infraestructura

La arquitectura de infraestructura por otro lado, abarca la suma de capacidades de tecnologías de información asociados a *hardware* y telecomunicaciones y esta asociado a la sinergia operacional y manejo de múltiples dispositivos que tomados juntos ofrecen un conjunto de servicios que en este caso particular sería la aplicación Aliadas.

En un principio las aplicaciones móviles eran mas sencillas, por lo que hacer la separación entre *front-end* y *back-end* no era necesaria redundando en que todo el código de la aplicación se ejecutaba en el dispositivo móvil. Sin embargo, mientras los recursos de hardware de los aparatos móviles han incrementado, también han incrementado las posibles funcionalidades que tienen los teléfonos móviles en el desarrollo de aplicaciones. Actualmente, las aplicaciones complejas tienen solo la capa de presentación equivalente a la arquitectura monolítica en el aparato móvil y consultan los datos a un *back-end* que se encuentra desplegado en internet. Debido a la complejidad de la aplicación a desarrollar en este proyecto, es sensato e incluso necesario desplegar esta lógica de negocios en internet y dejar la interfaz solamente funcionando en el aparato celular.

Al definir una arquitectura de infraestructura para aplicaciones móviles en la actualidad no existen demasiadas variantes al respecto a nivel conceptual, en otras palabras, independiente de si se utilizan tecnologías cloud u *on premise*, los conceptos a aplicar son los mismos, lo ideal es utilizar un *API Gateway* que consumirá todas las consultas provenientes del *front-end* de la aplicación y que a través de distintas rutas, redirija las consultas al servicio que contenga la funcionalidad a la que se está llamando, este se puede instanciar las veces que sea necesario para soportar la carga de consultas necesaria.

Esto cambia en el despliegue de servicios de *back-end*, ya que hay bastantes patrones a considerar al desplegar estos servicios según los recursos que se disponga [Richardson, 2019a].

Entre estos, la bibliografía sugiere alguno de los siguientes:

- Uso de un *host* para múltiples servicios:

  Este método consiste en la instalación directa de los paquetes necesarios en un *host* (como un servidor físico o una máquina virtual) para correr varios servicios las instancias que sean estimadamente necesarias. Esto conlleva a las siguientes ventajas y desventajas.

  - Ventajas:
    - Despliegue rápido de los servicios asumiendo el tener los paquetes instalados adecuadamente para correrlos.
    - Utilización eficiente de los recursos al aprovechar al máximo estos dentro del *host* al correr los servicios necesarios para levantar la aplicación.
  - Desventajas:

    - Riesgo de haber conflictos con el consumo de recursos de los servicios dentro el *host*.
    - El tener distintos servicios corriendo en un mismo *host* usando los paquetes instalados en este, pueden dar a lugar a conflictos de versión de estos paquetes (ejemplo: Servicio A requiere versión 2 de un lenguaje de programación y servicio B requiere versión 1).
    - Dificultad en limitar los recursos de cada servicio.

- Uso de máquinas virtuales en despliegue del servicio:

  Este método consiste en desplegar servicios empaquetados en imágenes de máquinas virtuales. Una imágen de una máquina virtual consiste en una imágen que una vez instanciada tiene todo lo necesario para poder correr un servicio (sistema operativo de máquina virtual, instalación de paquetes, lenguajes de programación, etc.), por ejemplo en *cloud* en un *Amazon Machine Image* que permite instanciar la cantidad de veces que sea necesaria una máquina virtual que corre el servicio desplegado. Esto conlleva a las siguientes ventajas y desventajas.

  - Ventajas:
    - La imagen de la máquina virtual encapsula el *stack* tecnológico, solucionando el problema de versiones de paquetes instalados del método anterior.
    - Las instancias de cada servicio están aisladas en cada máquina desplegada pudiendo asignar los recursos justos y necesarios para cada máquina virtual.
    - Uso de tecnologías *cloud* maduras y automatizadas que escalan la aplicación según sea necesario.
  - Desventajas:
    - Uso menos eficiente de la utilización de recursos, esto debido a la sobrecarga que implica levantar cada servicio en cada máquina virtual, incluyendo el uso de recursos en cada sistema operativo de cada máquina virtual instanciada.
    - Despliegues lentos, debido a que cada máquina virtual tiene que cargar además del servicio, el software necesario para levantar la máquina virtual.
    - Sobrecarga de administración del sistema, debido a que el encargado de las máquinas virtuales debe encargarse de actualizar el sistema operativo y lo necesario para levantar el servicio dentro de la máquina virtual.

- Uso de contenedores en despliegue del servicio:

  Los contenedores son un mecanismo a nivel de sistema operativo de virtualización que es más liviano que la solución ofrecida por ejemplo por máquinas virtuales. A través del empaquetamiento del código del servicio en una imagen, se crean contenedores que corren una o más instancias en una caja de arena que está aislada del ambiente del *host* donde está corriendo el contenedor excepto por los puertos/IP de la red en la que se encuentra el contenedor, permitiendo enlazar puertos a estos contenedores. El uso de este método presenta las siguientes ventajas y desventajas.

- Ventajas:
  Principalmente los mismos del despliegue de una máquina virtual
  - Encapsulamiento del stack tecnológico utilizado por el servicio.
  - Aislamiento de instancias del servicio.
  - Control sobre recursos asignados a cada servicio.

  Pero sin la sobrecarga de tener que utilizar un sistema operativo por cada contenedor instanciado, lo cual los hace más livianos.
- Desventajas:
  - Sin embargo si o si tiene que haber un encargado de mantener las imágenes de los servicios, actualizando el servicio que corre los contenedores a través de las imágenes.

Para aprovechar aún más el uso de los contenedores, también se pueden utilizar **orquestadores de contenedores**, lo cual permite utilizar un conjunto de máquinas que corren el servicio de contenedores como un pozo de recursos, permitiendo correr una cantidad especifica de instancias de cada servicio haciendo que el orquestador gestione los recursos para ello.

Un orquestador permite realizar 3 funciones principales:

- Manejo de recursos:
  A través del uso de un *cluster* de maquinas como un pozo de *CPU*, memoria y almacenamiento tratando el conjunto de máquinas como una sola.
- Planificación:
  Permitiendo seleccionar la máquina que correrá un contenedor en específico, lo cual considera los recursos requeridos por este.
- Manejo de servicios:
  La herramienta de orquestación de contenedores se asegura de que un número de instancias está corriendo de manera correcta a tiempo completo, además de ir actualizando los servicios y dar la opción de volver a versiones anteriores en casos necesarios.

- Despliegue *Serverless* del servicio:

  A través de un despliegue *serverless* de un servicio se delega la responsabilidad del despliegue de una aplicación a una nube pública teniendo que así no tener la necesidad de tener a alguien encargado de una infraestructura para desplegar los servicios necesarios para levantar la aplicación. Esto a través de funciones lambda que se pueden ejecutar y cobrar bajo demanda al que realice el despliegue. Una función lambda es bajo el contexto de este trabajo, un servicio sin estado que usualmente maneja péticiones invocando servicios de la nube. El precursor más grande actualmente de esta tecnología es AWS (*Amazon Web Services*) con AWS Lambda quien aparte ofrece un conjunto no menor de servicios para conectarlo con estas funciones en caso de ser necesario.

- Ventajas:
  - Está integrado con servicios que ya ofrece la *cloud* pública, lo cual facilita la conexión con persistencia y uso de *API Gateway* por ejemplo.
  - Elimina bastantes labores de administración de sistemas, por lo que se puede enfocar más recursos humanos a el desarrollo de aplicaciones.
  - Es elástico, osea, no hay que pensar en la cantidad de recursos necesarios para correr las funciones lambda implementadas, al menos a nivel de operaciones.
  - Precio basado en demanda de las peticiones.
- Desventajas:
  - Latencia debido a lo costoso en tiempo que puede ser correr una instancia de la aplicación que se desarrolló en lambda, por lo que esto no es compatible con peticiones que sean sensibles de tiempo.
  - Debido a la naturaleza extremadamente efímera de las funciones lambda es que no son adecuados para correr servicios de larga espera.

2.3. Metodologías para Requerimientos No-Funcionales

En lo que sigue, describimos los marcos teóricos y metodologías utilizadas para la definición de las políticas y mecanismos diseñados para satisfacer los requerimientos no-funcionales considerados en este trabajo derivados de [IIBA, 2015]

2.3.1. Seguridad

Para el tema de la seguridad hay que considerar que tanto la arquitectura de software como de infraestructura tienen formas distintas de abarcar este tema en particular en la arquitectura de software, se considera principalmente el entregar datos al usuario correcto según su identificador de usuario y/o rol dentro de la aplicación, sin embargo en la infraestructura, hay que considerar que las herramientas de comunicación principalmente no sean interceptadas, las políticas a implementar en la siguiente sección darán cuenta de esto señalando protocolos a utilizar al respecto.

2.3.2. Rendimiento

En las aplicaciones del *back-end*, el rendimiento puede verse mermado, por distintas razones, ya sea por mala optimización del código lo cual tendría que ver con la arquitectura de software o por falta de recursos, problemas de red o mal uso de protocolos dentro de la infraestructura, las políticas a implementar mitigaran estos problemas.

2.3.3. Mantenibilidad

La metodología de aplicación de 12 factores [Wiggins, 2017] facilita la mantenibilidad del software al definir buenas prácticas de desarrollo y despliegue de un sistema de software.

Para esto se sigue una lista de factores que se deben de cumplir y que serán evaluadas en la sección de ”Validación de la solución”. Los factores descritos a continuación se dividen en dos grupos. El primero, corresponde a aquellos que permiten definir políticas y mecanismos para la arquitectura de software, mientras que el segundo grupo corresponde a aquellos que lo permiten para la arquitectura de infraestructura.

- Arquitectura de Software:
    1. Código base:
       Que la aplicación tenga un código base quiere decir que debe estar dentro de un repositorio en un sistema de control de versiones (como **Git** por ejemplo), éste código es único, pero los despliegues de esta aplicación pueden ser múltiples según se necesite, el despliegue de producción es el más conocido pero también hay que tomar en consideración los despliegues locales en las máquinas individuales de los desarrolladores por ejemplo. Notar que la relación entre código base y despliegue es siempre de uno a muchos.
    2. Dependencias:
       Una aplicación de 12 factores debe de declarar explícitamente sus dependencias a través de un gestor de paquetes, ejemplos de esto es *bundler* en *Ruby*, *composer* en *PHP* y *pip* en *Python*. La aplicación debe de tener un manifiesto de los paquetes necesarios para correr la aplicación sin tener que recurrir a instalaciones explicitas dentro del sistema, estos manifiestos estan usualmente dentro de un archivo .json (composer.json por ejemplo en PHP).
    3. Configuraciones:
       La configuración en una aplicación es lo único que debe variar en una aplicación desarrollada bajo estos factores, ejemplos de configuraciones son direcciones para conexión de persistencia de datos, credenciales para servicios externos o entre otros.
       Una forma usual de hacerlo es a través del uso de variables de entorno que permiten cambiar estas configuraciones según donde esté desplegada la aplicación.
    4. *Backing Services*:
       Los *Backing Services* son recursos que la aplicación puede consumir a través de la red, ejemplos de estos son sistemas de persistencia de información (bases de datos) y servicios de correos con protocolo *SMTP* como **Postfix**. Estos servicios deben de ser accedidos a través de una URL que está almacenado en la **configuración** por lo que el servicio no tiene que hacer la distinción sobre si es un recurso obtenido localmente o de terceros (como por ejemplo consultar una base de datos en la nube).

5. Construcción, distribución y ejecución:

   Se debe realizar una separación entre fases de construcción, distribución y ejecución. Esto se logra utilizando las herramientas de despliegue gestionando las distribuciones como *releases* etiquetándolo con versiones o *timestamps* guardando un estado específico del código base permitiendo volver a este de ser necesario en caso de haber problemas con cambios nuevos desplegados. La ejecución de una aplicación se despliega en función de la construcción de la distribución junto a la configuración de donde se debe desplegar, en la sección ”validación de la solución”se mostrará un ejemplo claro de esto.

6. Procesos sin estado:

   Los procesos de las aplicaciones que cumplen con este estándar no deben de conservar ningún estado y cualquier información que requiera persistencia debe ser tratado con un *backing service*.

7. Igualdad entre desarrollo y producción:

   Las aplicaciones deben estar diseñadas para realizar despliegues continuos, osea, tienen que disminuir las diferencias entre entorno de desarrollo de producción lo más posible. Este esta relacionado a la arquitectura de software.

8. Historiales:

   Las aplicaciones no se preocupan del almacenamiento del historial (o *logs*) en un sistema de archivos, solamente escribe los eventos ocurridos vía salida estándar, permitiendo ver por terminal en caso de ser necesario lo que ocurre con la interacción de la aplicación. Este esta relacionado a la arquitectura de software.

9. Administración de procesos:

   Las aplicaciones deben proporcionar una consola tipo **REPL** (*read-eval-print loop*) que permita realizar operaciones de administración o mantenimiento de forma poco recurrente, por ejemplo ejecutando migraciones para una base de datos desde esta consola para poder utilizar sus tablas. Este esta relacionado a la arquitectura de software.

- Arquitectura de Infraestructura:

1. Asignación de puertos:

   Las aplicaciones pueden ser expuestas a través del uso de puertos para ser consultados a través de estos sin la necesidad de un servidor web. Este esta relacionado a la arquitectura de infraestructura.

2. Concurrencia:

   La aplicación que cumple con estos estándares, debe presentar facilidades para la asignación de recursos a sus respectivos procesos permitiendo escalar horizontalmente si es necesario además de poder ser utilizados por un gestor de procesos que facilite su despliegue, historial e información sobre término del proceso y automatización del reinicio en caso de la caída de estos. Este esta relacionado a la arquitectura de infraestructura.

3. Desechabilidad:

   Una aplicación puede inicializarse o finalizarse en el momento que sea necesario sin consecuencias adversas, lo cual entrega robustez al servicio. Este esta relacionado a la arquitectura de infraestructura.

2.3.4. Escalabilidad

La manera en que se puede escalar una aplicación depende principalmente de la arquitectura de software en la que esta esté montada. Sin embargo, podemos identificar 3 formas principales de escalar una aplicación, las que se encuentran determinadas por el cubo de escalabilidad en [Abbott M. L., 2010] representadas por los ejes de coordenadas que representan las dimensiones de este cubo, cada uno con sus respectivos desafíos a implementar. La figura 1 ilustra la interacción entre los 3 ejes descritos a continuación.

- Eje X (horizontal), Replicación de instancias:

  Escalar en el eje X consiste en replicar instancias de una aplicación o una base de datos (por ejemplo, con un balanceador de carga), donde cada instancia maneja $\frac{1}{N}$ de la carga recibida donde N es la cantidad de instancias duplicadas.

- Eje Y (vertical), División de datos heterogéneos:

  Escalar en el eje Y consiste en descomponer funcionalmente la aplicación en distintos módulos o servicios (macro y micro) lo que permite escalar cada servicio independientemente y administrar recursos a los servicios en función de lo que estos requieran.

- Eje Z, División de datos homogéneos:

  Escalar en el eje Z consiste en particionar los índices de la base de datos considerando criterios de similaridad entre las entradas que estos indexan (por ejemplo, la base de datos A se encarga de gestionar clientes “VIP”, mientras que la base de datos B, “clientes normales”).

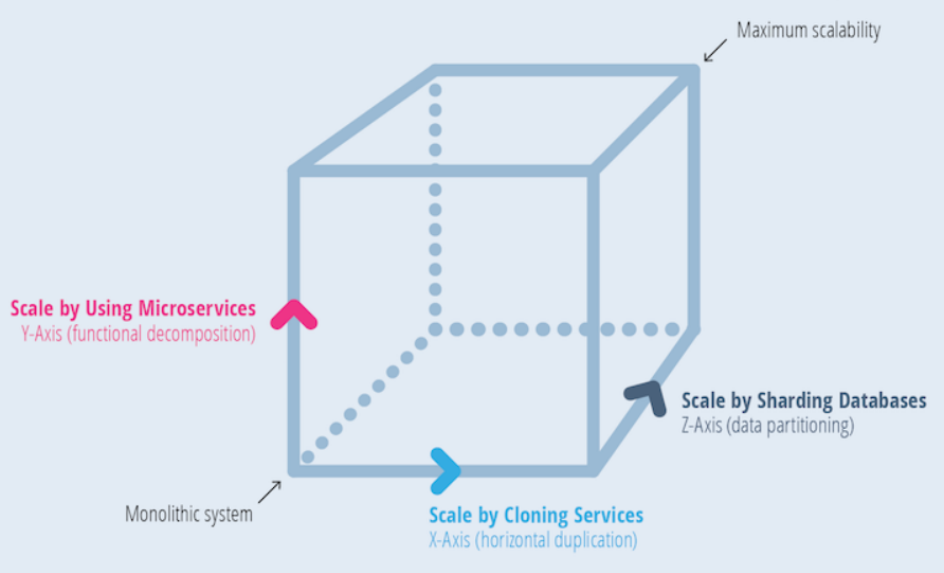


Figura 7: The Scale cube and Microservices: 3 dimensions to Scaling.
Fuente: `https://thenewstack.io/from-monolith-to-microservices/`

Idealmente, la arquitectura de una aplicación debe poder escalar fácilmente en al menos dos de los ejes recién descritos. Al mismo tiempo, la arquitectura de software debe permitir que el escalamiento ocurra interrumpiendo lo menos posible el funcionamiento de los servicios provistos por la aplicación cuando esta se encuentre en producción.

El cumplimiento de estas políticas asociadas a los requisitos no funcionales, justificará el uso de las herramientas, servicios y componentes que se utilizarán en el presente proyecto para diseñar e implementar la arquitectura de software e infraestructura sobre la que funcionará la aplicación Aliadas.

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

3.1. Diseño de Arquitectura de Software Propuesta

Para el diseño de la arquitectura de software, se utilizaron los requisitos que fueron discutidos con el equipo de diseño, como se mencionó en la definición del problema. Se decidió diseñar una arquitectura de software basada principalmente en *microservicios* esto debido a que la división de la lógica de negocios en servicios va a permitir a corto y a largo plazo mayor escalabilidad considerando que una de las premisas de la aplicación Aliadas es recibir potencialmente gran demanda, por lo que la facilidad de escalabilidad que ofrece una arquitectura de microservcios es ideal para enfrentar este caso.

3.1.1. Beneficios y desafíos

Como se dijo anteriormente, la arquitectura será basada principalmente en microservicios para obtener los siguientes beneficios [Richardson, 2019b]

- Permite despliegue y entrega continua de aplicaciones complejas y grandes.
- Los servicios (si están bien diseñados) son pequeños y fáciles de mantener.
- Cada servicio se despliega independientemente.
- Cada servicio es independientemente escalable.
- La arquitectura permite que los equipos de desarrollo de cada servicio sean autónomos.
- Permite facilidad para experimentar y adoptar nuevas tecnologías.
- Se tiene mejor aislamiento de fallas, pues al separar los servicios se puede saber inmediatamente en qué servicio está el problema.

Sin embargo, la implementación de una arquitectura basada en microservicios tiene los siguientes desafíos para su implementación:

- La definición de servicios es compleja: debido a que se debe de conocer bien el dominio y la interacción entre actores y entes dentro del contexto de la aplicación para granular los servicios.

- El despliegue requiere de conocimientos de sistemas distribuidos, lo cual vuelve complejo el despliegue, las pruebas y el desarrollo de la aplicación.
- Desplegar características que requieran el uso conjunto de distintos servicios requiere coordinación de servicios.

Por lo tanto, se debe de usar algún patrón de división de servicios cuya forma en la que se dispuso a realizar fue usando el patrón de descomposición de servicios por capacidad de negocio [Richardson, 2021] lo cual propone servicios que tengan una relación 1 a 1 entre capacidad de negocio y servicio. Una capacidad de negocio es un atributo del negocio que le añade valor a este. Se dará un ejemplo después de identificar los componentes y servicios. Aparte, se debe de obtener los conocimientos sobre sistemas distribuidos y por último utilizar herramientas que simplifiquen la conexión entre servicios y aquí es donde entra el uso de la arquitectura basada en eventos para la conexión de servicios que se explicará en la elección de tecnologías.

3.1.2. Servicios y Componentes Identificados

A continuación se describirán más específicamente los servicios y componentes identificados para el funcionamiento de la aplicación y la razón de su uso. Se debe hacer énfasis no obstante, en la diferencia entre componente y servicio dentro del contexto de esta memoria, siendo un componente una solución existente que tome el papel de un servicio dentro del sistema, por ejemplo, como se verá más tarde el componente Keycloak tiene el rol de servicio de Proveedor de identidad dentro del conjunto de servicios de la arquitectura.

En general, se utilizó la política de implementar soluciones *open source* como componentes cuando fuera posible, esto debido a la facilidad de mantener estas aplicaciones comparado a hacer código desde cero para tal servicio.

Para el desarrollo de servicios desde cero se decidió tomar como preferencia el uso del lenguaje de programación **Python** debido a que tarde o temprano se debía usar para implementar el servicio del sistema recomendador para la aplicación que responderá solicitudes de larga duración por la naturaleza de los datos a guardar de la aplicación (esto es un trabajo ajeno al de este trabajo de título), que es un servicio crítico para la aplicación. Adicionalmente, *Python* posee librerías que ayudan bastante a desarrollar microservicios, es por esto que se decidió utilizar **FastAPI** que es un *framework* diseñado para desarrollar microservicios.

Como último recurso se considerará utilizar otros lenguajes y/o tecnologías para servicios específicos.

- *API Gateway*:

Para el *API Gateway* se utilizó el componente **Kong** que es un servicio de API Gateway que se encarga de dar una capa de abstracción entre la capa de presentación y los servicios desplegados de la aplicación.

Se utilizó esta tecnología debido a la facilidad que otorga añadir rutas que redirijan a los respectivos servicios y además de que ofrece la opción de utilizar *plugins* que pueden servir para la aplicación en caso de ser necesario y contar con balanceador de carga.

- Proveedor de identidad:

  Para el servicio de proveedor de identidad se utilizó el componente **Keycloak** sistema *open source* de manejo y acceso de identidad para aplicaciones y servicios.

  Se decidió utilizar debido a que es una de las soluciones que usa Red Hat y porque presenta soluciones para los estándares usuales de autenticación como *OAuth 2* y *Open Id Connect*. Además, permite extender funcionalidades de ser necesario.

- E-commerce:

  Para el servicio de *e-commerce* se utilizó **Spree** que es un sistema *open source* implementado en **Ruby on Rails**, este sistema es un monolito al cubrir gran parte de los problemas del dominio de la aplicación como por ejemplo, aparte del manejo de productos y vendedores, el manejo de taxonomías (categorías) e imágenes. Solo se utiliza el servicio de API REST de la aplicación más conocido como “headless”.

  La decisión del uso de esta tecnología va en que debido al tiempo de desarrollo que se tiene propuesto para el proyecto, no se cuenta con el tiempo para hacer un sistema *e-commerce* desde el principio basado en microservicios con los requisitos asociados. Por lo que fue necesario utilizar alguna aplicación que ya tenga solucionado el dominio del *e-commerce* y que de la opción de extender sus funcionalidades según los requisitos de la aplicación.

  Aún así que se tiene claro que efectivamente se puede dividir en más servicios.

- Búsqueda:

  Para el servicio de búsqueda se utiliza **Elasticsearch** el cual genera índices a partir de los datos guardados en las bases de datos de las aplicaciones, en particular ahora se utiliza para facilitar la búsqueda de tiendas y productos.

  Se utiliza esta tecnología por la fácil adaptación que presenta con **Spree** y por las herramientas que ofrece para facilitar las búsquedas dentro de la aplicación además de asegurar de que la búsqueda funcione de mejor forma desacoplándola de la aplicación e instanciar con hardware dedicado

- Chat:

  Para el servicio de *Chat* se utilizará el componente **Rocket Chat**, esto debido a la facilidad que tiene para conectarse con **Keycloak** y a proveedores de identidad en general, además de abarcar los casos de uso que tendrá la aplicación para habilitar comunicación entre los usuarios dentro de la aplicación.

- Registro de Acciones:

  Este servicio se encarga de capturar las acciones de los usuarios, ya sean por búsqueda, visita o intención de compra de algún producto o tienda cuando aplique, este servicio fue desarrollado en **Fast Api**, un framework desarrollado en Python para micro servicios y se guarda la data en MongoDB.

- Captura de productos:

  Este servicio se encarga de importar y exportar productos desde otros sistemas de e-commerce, entre ellos están Mercado Libre, Facebook e Instagram, Idealmente utilizando las herramientas de *API* que ofrecen estos sistemas.

- Sistema de recomendación:

  Este servicio se encarga de consumir la *API* de Registro de acciones y Elasticsearch para reordenar los resultados de búsqueda según las acciones del usuario para sugerir productos adecuados según lo consumido. Este reordenamiento favorece la diversidad de los resultados, esto con el fin de generar una mayor visibilidad de los productos y vendedores dentro de la aplicación.

- Manejador de *API* de administrador de proveedor de identidad:

  Este servicio permite abstraer las consultas de la *API* de administrador para que no estén todas expuestas al público, exponiendo y restringiendo solo las consultas que sean necesarias para el funcionamiento de la aplicación Aliadas.

  Para esto se utilizaron distintos flujos del protocolo *OpenID Connect*, los cuales se explicarán brevemente en la sección de seguridad dentro de esta misma sección.

Un ejemplo sobre la capacidad de negocio relacionada a los servicios definidos, anteriormente, es la relación entre el servicio *e-commerce*, de registro de acciones y el sistema recomendador. En el caso de esta aplicación el registro de acciones registra las visitas e intenciones de compra de los productos del servicio *e-commerce*, estas acciones son consultadas por el sistema recomendador para posteriormente recomendar al usuario productos en función de sus acciones, otorgando valor a la lógica de negocios de la aplicación Aliadas.

Notar que se utilizaron tecnologías *open source* en algunos casos y en caso de tener que implementar funcionalidades específicas, se desarrollaron desde 0, aún así en algunos casos se tuvo que añadir u adaptar código en algunos de estos proyectos para adaptarlos a las necesidades de la aplicación, estos servicios son:

- Proveedor de Identidad: Donde se tuvo que añadir código para poder consumir el sistema de autenticación de **Mercado Libre** y traducir algunas secciones que estaban en Inglés al Español.
- *E-commerce*: Donde se tuvieron que añadir entre muchas cosas, atributos a los usuarios extendiendo los modelos que vienen con **Spree**, hacer la conexión con el servicio de búsqueda de **Elasticsearch**, métodos, entre otros.

3.1.3. Arquitectura Propuesta

En la figura 8 se muestran un diagrama de las funcionalidades principales resumidas de los servicios

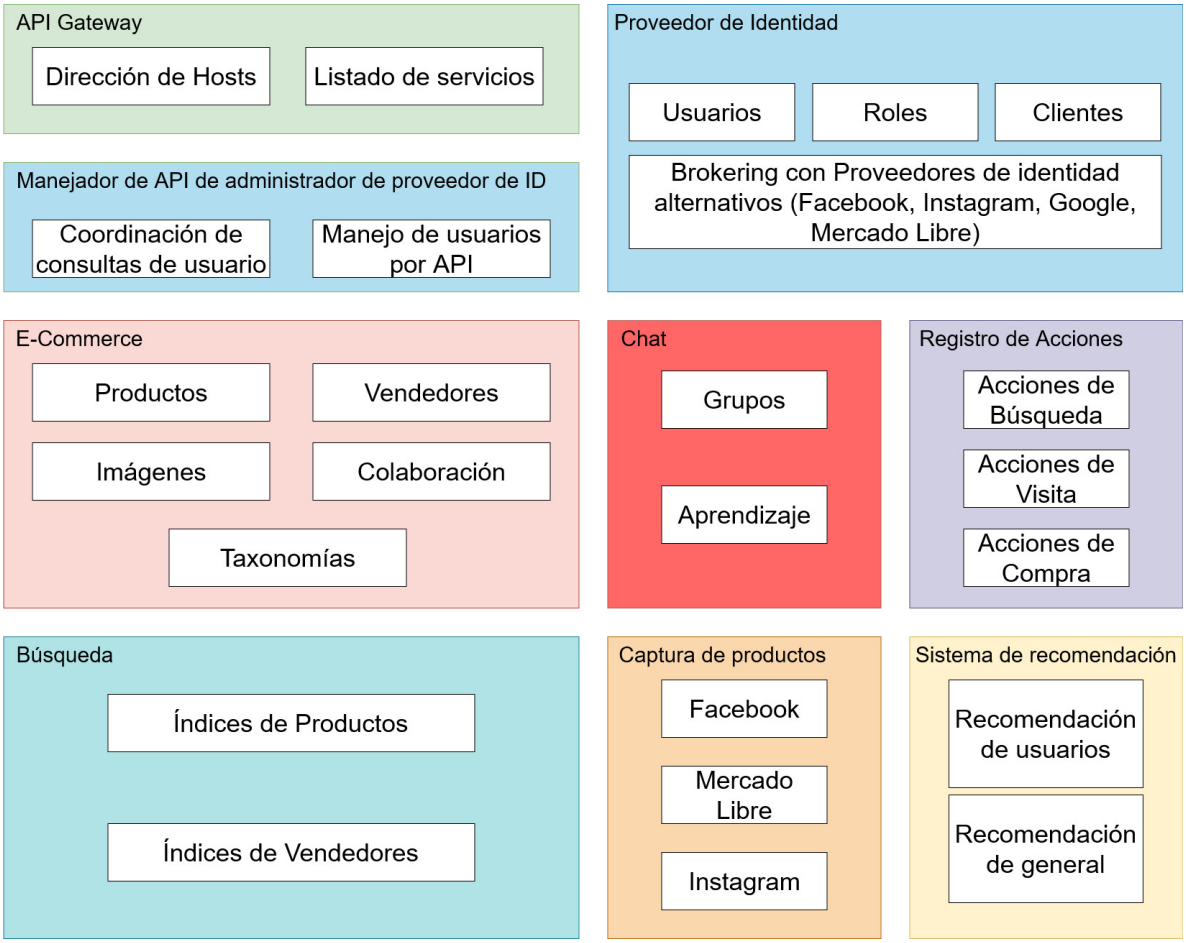


Figura 8: Ilustración de servicios desplegados con sus funcionalidades principales
Fuente: Elaboración propia

Por otro lado, se mostrará con un ejemplo el funcionamiento a grandes rasgos de la arquitectura de software, en la figura 9 se muestra el flujo por le que pasa la petición de datos para una ruta del sistema recomendador, en el paso 1 se inicia sesión con el proveedor de identidad (los detalles del protocolo de autenticación son tópico para la sección siguiente), posteriormente, se hace la petición de los datos a la *API Gateway*, luego el API Gateway, según la ruta a la que se llama, deriva la consulta al servicio correspondiente en el paso 3, en este caso, al sistema recomendador, luego, en el paso 4, llega la consulta al sistema recomendador y realiza las interacciones necesarias con los otros servicios para devolver la recomendación a la API Gateway, para posteriormente responder con los datos en el paso 6 al cliente de interfaz de usuario.

Este flujo es parecido para las consultas hacia todos los servicios.

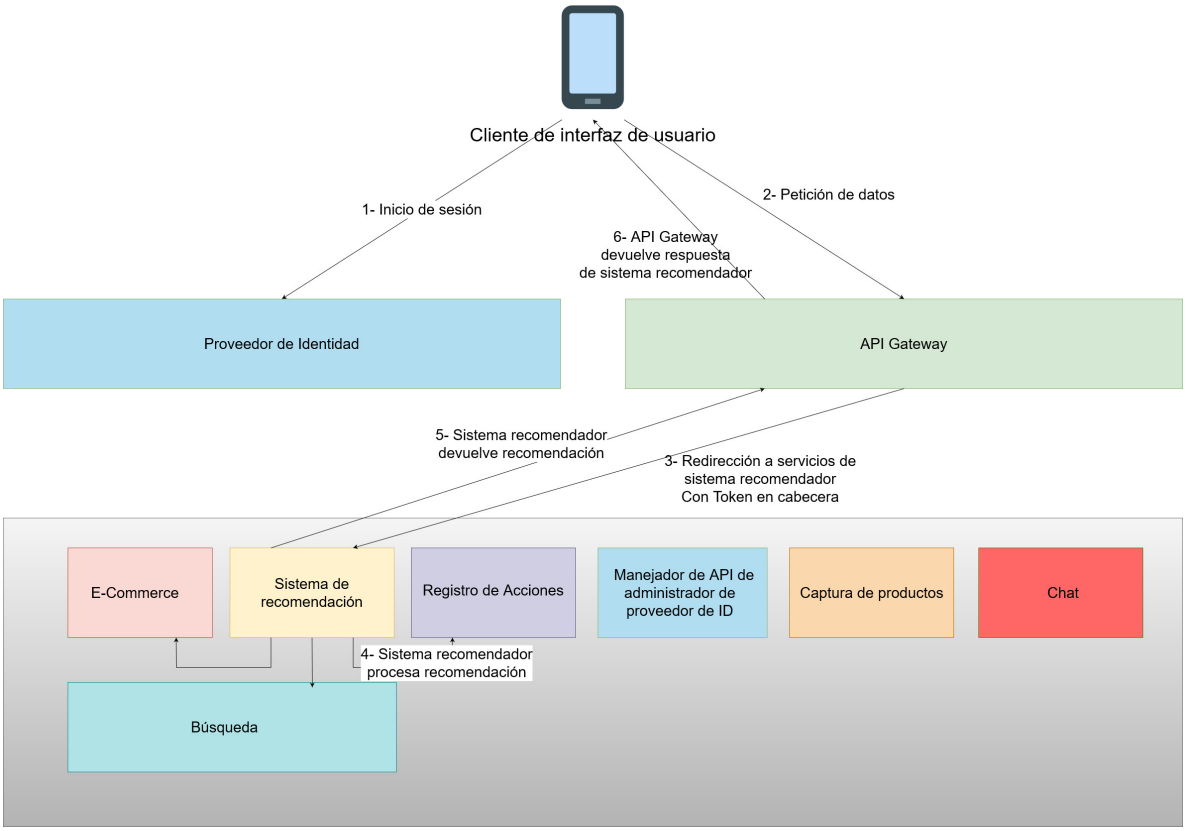


Figura 9: Ejemplo de interacción de cliente de interfaz con sistema recomendador
Fuente: Elaboración propia

3.2. Diseño de Arquitectura de Infraestructura Propuesta

Ya que se tiene definida la arquitectura de software, se puede hablar de definir una arquitectura de infraestructura, para esto se utilizó el patrón de *uso de contenedores en despliegue del servicio*, esto principal debido a lo complicado que sería levantar servicios en un terminal y tener que instalar compiladores e interpretes de lenguajes de programación y gestores de paquetes de estos sin el uso de contenedores que tendrían su propio ambiente ideal para desplegar cada uno de los servicios con sus propias características y requisitos técnicos.

3.2.1. Hardware utilizado

Para lograr esto se cuenta con varias Máquinas virtuales para correr los servicios de Kong, Keycloak, Mongo y Elasticsearch y un servidor para correr los servicios de la aplicación en

contenedores.

Nombre de terminal	Procesador	Memoria RAM	Almacenamiento	Sistema operativo
Máquina virtual Keycloak	Intel Xeon E312xx (Sandy Bridge), 4 núcleos	4 GB	20 GB	RHEL 8
Máquina virtual Kong	Intel Xeon E312xx (Sandy Bridge), 4 núcleos	4 GB	20 GB	RHEL 8
Máquinas virtuales para MongoDB	Intel Xeon E312xx (Sandy Bridge), 2 núcleos	4 GB	100 GB	RHEL 7
Máquinas virtuales para Elasticsearch	Intel Xeon E312xx (Sandy Bridge), 2 núcleos	4 GB	100 GB	RHEL 7
Máquinas virtuales para RabbitMQ	Intel Xeon E312xx (Sandy Bridge), 2 núcleos	4 GB	100 GB	RHEL 7
Servidor de despliegue de servicios de la aplicación	Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz, 6 núcleos	32 GB	6 TB	RHEL 7

Tabla 1: Tabla descriptiva de recursos computacionales para la infraestructura
Fuente: Elaboración Propia.

3.2.2. Mapping

La infraestructura como está en este instante y siguiendo el contenido de la 1, provee de 1 máquina virtual dedicada para el *API Gateway*, 1 para **Keycloak**, 1 para **Elasticsearch**, 1 para **RabbitMQ**, 1 para **MongoDB** y el resto de los servicios está siendo ejecutado en el servidor de despliegue de servicios de la aplicación, todos con contenedores, excepto la *API Gateway* y los *backing services* que serían **Elasticsearch**, **RabbitMQ** y **MongoDB**, por otro lado existe una tecnología *cloud* de almacenamiento que es **Amazon S3**, el único *backing service* que no tiene su propia máquina virtual es **PostgreSQL** que está desplegado en el servidor de despliegue de servicios de la aplicación.

3.2.3. Arquitectura de Infraestructura Propuesta

Se muestra la siguiente ilustración que muestra gráficamente la arquitectura de infraestructura para satisfacer las necesidades de los futuros clientes de la aplicación [3]. En la figura 10 se muestran los dispositivos y actores asociados con la arquitectura de la aplicación.

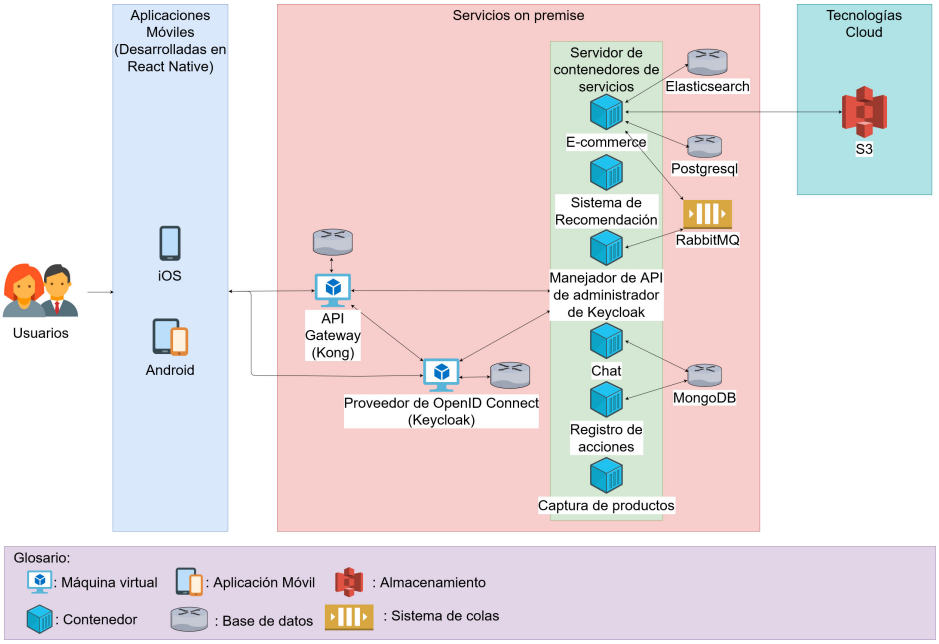


Figura 10: Ilustración de arquitectura propuesta
Fuente: Elaboración propia

Se muestra los despliegues de los servicios y su conexión a los *backing services* como las bases de datos y el sistema de colas, notar que el servidor de contenedores de servicios tiene todos los servicios corriendo excepto el *API Gateway* y **Keycloak** que están en sus respectivas máquinas virtuales pero aún así **Keycloak** esta containerizado también dentro de esa máquina virtual, las bases de datos de la *API Gateway* y **Keycloak** están dentro de las máquinas virtuales mismas.

3.2.4. Despliegue

Recordar que el despliegue de las aplicaciones esta realizado con contenedores, específicamente, con **Docker** y **Docker Compose**, con **Docker Compose** se puden definir y orquestar distintos servicios a través de un archivo con formato '.yml'.

[3] Los esfuerzos en este trabajo de título se enfocarán principalmente en habilitar la sección roja (Red de la aplicación) mostrada en la figura 10

Por lo general se definen dos archivos '.yml' para el despliegue, uno para despliegues en ambientes de desarrollo que tienden a tener todos los *backing services* necesarios para hacer funcionar el servicio y facilitar el desarrollo de la aplicación a los desarrolladores y otro para ambientes de producción que tienden a tener solo 1 o a lo más 2 despliegues, a continuación se mostrará un ejemplo de esto, en particular, un buen ejemplo es el del servicio de *e-commerce*, en el siguiente enlace `https://pastebin.com/kbp4xx2E` se muestra el archivo de despliegue de desarrollo, como se puede notar, dentro del archivo se definen 5 despliegues, donde por orden son, una instancia de **elasticsearch**, una base de datos **PostgreSQL**, el mismo despliegue del servicio *e-commerce*, un consumidor de colas, sobre lo cual se explicará en la sección de elección de tecnologías, y **Kibana** que es una interfaz para explorar los datos guardados por **Elasticsearch** para facilitar su consulta.

Cada uno de estos despliegues tiene sus propias variables de entorno y credenciales por defecto, pues la intención es facilitar el desarrollo desplegando todos esos servicios y *backing services* sin casi ninguna configuración.

Por otro lado, se tiene otro archivo que solo corre el servicio en si y el consumidor de cola (enlace aquí `https://pastebin.com/NExAqPj1`), esto se realiza porque la intención de este archivo es utilizarlo para cuando se despliegue en la misma infraestructura que ya tiene corriendo los *backing services* dentro de ella, por lo que no hay necesidad de desplegar nuevamente esos *backing services*.

Por último, estos archivos que utiliza **Docker Compose** utilizan un **Dockerfile**, el cual tiene todos los pasos para poder desplegar el servicio en este caso de *e-commerce* en especifico

```
FROM ruby:2.7.4

RUN apt-get update && apt-get install -y nodejs postgresql-client \
&& apt-get install -y \
curl \
build-essential \
libpq-dev && \
curl -sL "https://deb.nodesource.com/setup_10.x" | bash - && \
curl -sS "https://dl.yarnpkg.com/debian/pubkey.gpg" | apt-key add - && \
echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list && \
apt-get update && apt-get install -y nodejs yarn

RUN mkdir /myapp
WORKDIR /myapp

COPY Gemfile /myapp/Gemfile
COPY Gemfile.lock /myapp/Gemfile.lock

RUN bundle install
RUN bundle update --all

COPY . /myapp

EXPOSE 3000

COPY package.json /myapp/package.json
COPY yarn.lock /myapp/yarn.lock
RUN yarn install --check-files
```

Figura 11: Dockerfile detallando pasos de despliegue en la generación de la imagen en la que estará basado el contenedor instanciado

Fuente: Elaboración propia.

3.3. Políticas y Mecanismos Propuestos para RNF

3.3.1. Seguridad

Los políticas de seguridad tienden a depender netamente del tipo de aplicación que se está desarrollando y la infraestructura que se este usando, como se esta desarrollando un aplicación basada en *marketplace e-commerce* podemos definir las siguientes políticas de seguridad que van a ser implementadas en la arquitectura de software e infraestructura cuando corresponda.

Una de las tecnologías en despliegue de API de *e-commerce* de nombre **Fabric** [Kanjilal, 2021] , subió un articulo con una lista de políticas recomendadas al desarrollar una aplicación *e-commerce* .

Sin embargo, se omitirán algunas, y se modificarán levemente otras debido a que la aplicación Aliadas, no manejara pagos, sin embargo se mencionarán aparte para considerar en caso de que se decida a futuro implementar.

- Para arquitectura de software:
  - Mantener librerías, *frameworks* y plataformas actualizadas.
  - Uso de protocolos de autenticación validados que permitan acceder a los usuarios a los recursos que netamente le correspondan según su rol y identificador de usuario.
- Para arquitectura de infraestructura:
  - Uso de HTTPS y Certificados SSL (*Secure Socket Layer*), debido a que brindan encriptación de los datos de punto a punto lo cual sirve bastante cuando se esta autenticando usuarios. (Se explica en mayor detalle el protocolo SSL en la elección de tecnologías)
  - Las consultas desde los clientes de interfaz de usuario deben de solo comunicarse a través del *API Gateway* a los servicios desplegados.

Políticas relacionadas a sistemas de pago que pueden servir a futuro si se implementan en la aplicación Aliadas:

- Evitar guardar datos sensibles, como por ejemplo datos de tarjetas de crédito y datos bancarios.
- Uso de procesadores de pago de confianza.
- Entender y cumplir con requerimientos PCI-PSS (*Payment Card Industry Data Security Standard*), que es un comité conformado por las compañías de tarjetas de crédito y débito para establecer prácticas de seguridad en la manipulación de estos.

3.3.2. Rendimiento

En rendimiento se aplicarán políticas que ayudarán a mejorar las consultas a la infraestructura desde la interfaz de usuario de Aliadas.

- Para arquitectura de software:
  1. Uso de Cache en servicios.
- Para arquitectura de infraestructura:
  1. Uso de soluciones de balanceo de carga cuando sea necesario.
  2. Indexación apropiada de bases de datos.

3.3.3. Mantenibilidad

En el tema de mantenibilidad, lo que hay que asegurar son buenas prácticas que aseguren que se cumplan las reglas de los 12 factores [Wiggins, 2017].

Estas se resumen en:

- Para arquitectura de software:
    1. Definir bien variables de entorno para despliegues de desarrollo
    2. Evitar acoplar direcciones de *backing services*
    3. Evitar guardar estados dentro de estos servicios, pues para ello están los backing services.
    4. Promover uso de historiales para ayudar a encontrar fallas.

3.3.4. Escalabilidad

- Para arquitectura de software: En el tema de escalabilidad, según el cubo de escalabilidad mencionado en el marco teórico [Abbott M. L., 2010], idealmente, la arquitectura de una aplicación debe poder escalar fácilmente en al menos dos de los ejes descritos en la bibliografía. Al mismo tiempo, la arquitectura de software debe permitir que el escalamiento ocurra interrumpiendo lo menos posible el funcionamiento de los servicios provistos por la aplicación cuando esta se encuentre en producción.

3.4. Plan de contingencia

El día de hoy se realizan respaldos una vez al día a las 0 horas de las bases de datos de los servicios, estos se guardan en un servidor NFS (*Network File System*) ubicado en el servidor de contenedores de servicios, por lo que en caso de corrupción severa de datos en la persistencia, se cuente con eso para volver a ese estado. Otra opción válida y que se podría hacer a futuro es guardar los respaldos de persistencia no en el servidor NFS sino que en un sistema de almacenamiento de archivos como lo es **Amazon S3**

En caso de que se deje de tener al datacenter del departamento de informática como proveedor de infraestructura, una idea es utilizar servicios *cloud* en casos de emergencia, sin embargo por cada mes se estaría pagando las instancias las cuales no se cobran por demanda, sino que por recursos por hora, como por ejemplo si se utilizan máquinas virtuales con **Amazon EC2**, sin embargo si se tuviera dinero, esta sería una opción a considerar para estos casos.

3.5. Elección de las Tecnologías

En esta sección se explicará el proceso que se siguió para decidir las tecnologías a utilizar en el proyecto. Para la elección de los componentes utilizados, se decidió optar principalmente a tecnologías *open source* que abarquen soluciones que ayuden al desarrollo de la aplicación móvil Aliadas, para ello se utilizaron tecnologías para abarcar las siguientes soluciones asociadas al proyecto sobre los cuales se tuvo que decidir dentro de un puñado de estos que se investigó.

- *Frameworks* de desarrollo de *APIs*:

  Si bien existen bastantes propuestas a utilizar, se consideraron 3 tecnologías principalmente al seleccionar un *framework* de desarrollo de *APIs*, esto en función principalmente de los conocimientos de los desarrolladores de la aplicación en el lenguajes de programación asociado a la tecnología.

  Estas tecnologías son **Lumen**, **Flask** y **FastAPI**

Tecnología	Lenguaje de programación	Documentación Automática	Librerías de manejo de vectores
Lumen	PHP	Sí, pero no por defecto	Sí pero con características limitadas
Flask	Python	Sí, pero no por defecto	Sí
FastAPI	Python	Sí, integrado por defecto	Sí

Tabla 2: Tabla descriptiva de tecnologías de *frameworks* de *APIs*
Fuente: Elaboración Propia.

Entre estos 3 se eligió **FastAPI** por 2 razones principales, primero, estaba hecho en **Python** y uno de los servicios desarrollados, específicamente el sistema recomendador, necesita librerías para manejar matrices y vectores junto a procesamiento de texto, para lo cual los desarrolladores involucrados tenían más experiencia en el uso del lenguaje **Python**, y segundo, tiene integración nativa con **OpenAPI 3.0** lo cual permite documentar automáticamente lo desarrollado en la aplicación, en la figura 12 se muestra un ejemplo de ello, en particular de 3 rutas con su respectiva sección explicativa de como funciona esa ruta de la *API*.

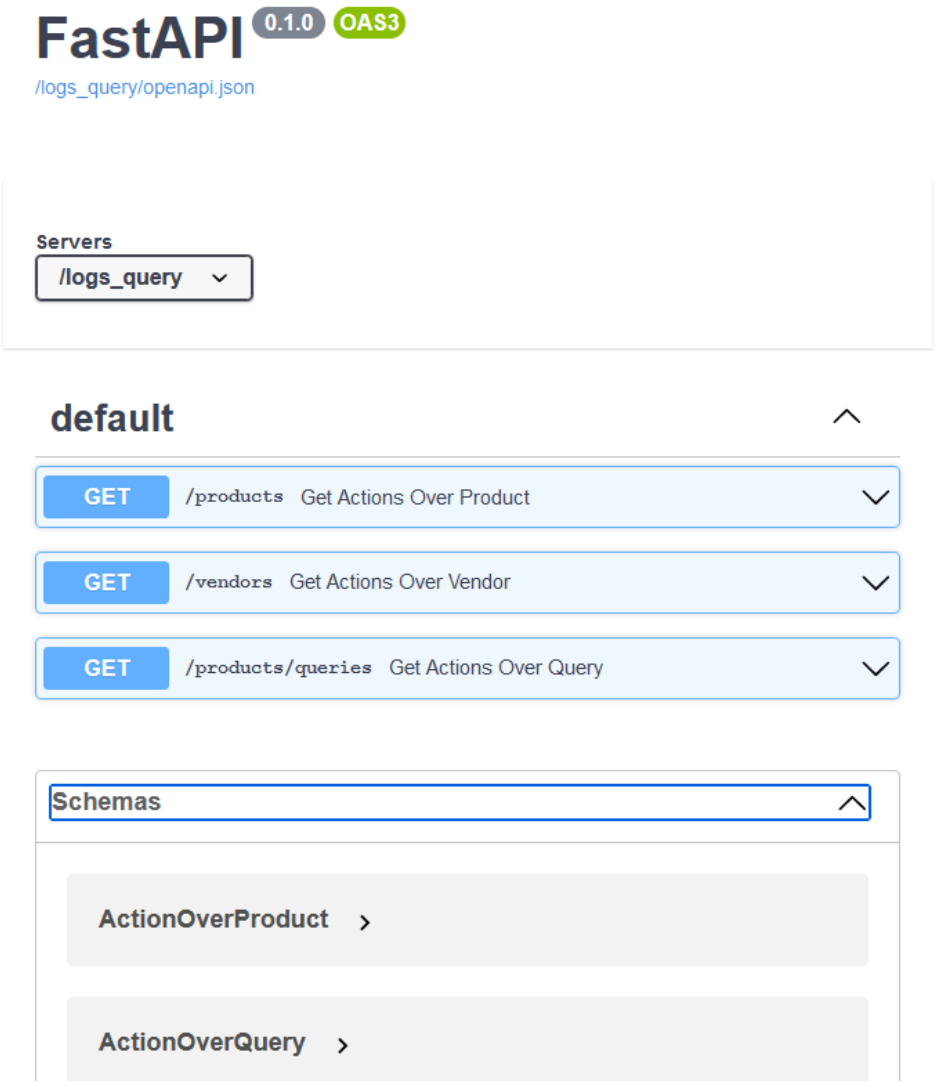


Figura 12: Ejemplo de documentación generada por **OpenAPI** en el despliegue del servicio de registro de acciones de usuario
Fuente: Despliegue local de servicio de registro de acciones

Por otro lado, **FastAPI** ofrece un método de paralelismo que calza con la infraestructura que se esta utilizando, que es el uso de *workers* que permiten instanciar varios procesos dentro de un servidor sacando ventaja de los núcleos del procesador para esto, paralelizando en este caso las consultas recibidas por los clientes móviles, en anexos se adjuntará la documentación al respecto.

- Soluciones para parte de dominio *e-commerce* de la aplicación:

  En un principio se tuvo la duda de si implementar desde el principio una solución *e-commerce* basada en microservicios desarrollada por el mismo equipo, sin embargo,

se llegó a la conclusión de que hacer esto es demasiado trabajo para la duración del proyecto y para las personas que estaban involucradas en el desarrollo. Por lo tanto se prefirió utilizar una solución *open source*

Esto fue una de las soluciones más difíciles de encontrar para la aplicación Aliadas, debido a que cada aplicación *e-commerce* está enfocado en sus propios requisitos funcionales, se probaron bastantes sin mucho éxito hasta que encontramos uno que era lo más parecido a lo que calzaba con nuestros requisitos.

Se hizo necesario utilizar una solución *e-commerce headless* es decir, una aplicación que no tenga acoplada una capa de presentación necesariamente y que permita ser utilizada por otra capa de presentación, en este caso desarrollado por el grupo de *front-end* del equipo de desarrollo de Aliadas.

Las soluciones que se analizaron fueron principalmente los siguientes además se muestran las funcionalidades que se necesitan en la aplicación Aliadas y si es que estas tecnologías las poseen por defecto o no.

Tecnología	Lenguaje utilizado y *framework*	Función de *Marketplace*	*API Headless*	Desacoplado de vistas
NopCommerce	.net basado en framework ASP.net	No	Sí, con plugin	No
Shuup	Python basado en *framework* Django	Si, pero pagado	Si, pero pagado	No
Vendure	Javascript, Typescript basado en *framework* Node.js	No	Si	Si
Spree	Ruby basado en *framework* Ruby on Rails	Si, con *plugin*	Si	Si

Tabla 3: Tabla descriptiva de tecnologías de *e-commerce* probados para acoplar al conjunto de servicios de Aliadas

Fuente: Elaboración Propia.

Se intentó primero utilizar **nopCommerce** que si bien tiene una extensión para usar como *e-commerce headless* era bastante limitado y el grupo de desarrolladores no poseía conocimientos de .net por lo que se descartó esta tecnología aparte de que integrar la lógica de *marketplace* había que hacerlo desde el principio.

Considerando la experiencia de los miembros del equipo se consideró después probar con **Shuup** que está basado en el lenguaje Python sin embargo no se logró desplegar adecuadamente y las funciones que necesitábamos requerían pago o asesoramiento pagado.

Posteriormente se intentó utilizar **Vendure**, basado en el *framework* **Node.js** el cual si bien cumple con lo requerido excepto por la implementación de marketplace, esta tecnología estaba en fase beta en la fecha en que se empezó a desarrollar el proyecto Aliadas por lo que decidimos descartarlo por tener potenciales problemas.

Al final se decidió por utilizar **Spree** debido a que aparte de que esta tecnología cumplía con lo que se necesitaba para el proyecto Aliadas, este permitía desacoplar las vistas de *e-commerce* por defecto permitiendo consumir los datos de la aplicación *e-commerce* desde cualquier aplicación *front-end* que se desarrolle como la aplicación móvil de Aliadas.

Notar aún así que al ser tecnologías *open source*, el *marketplace* se puede implementar desde 0 modificando el código base de la tecnología *open source* pero se consideró que esa parte del dominio era muy compleja para poder desarrollarlo a tiempo en la duración del proyecto, sin embargo en la investigación se encontró una extensión de **Spree** que añade la capacidad de agregar tiendas, por lo que con eso ya tendriamos cubierto el problema de la implementación del dominio de *e-commerce*.

Además, **Ruby on Rails** tiene su propia solución para implementar *workers* llamada **Puma**.

- Autenticación y autorización de usuarios:

  La autenticación y autorización de usuarios a recursos de los servicios es una problemática que es factor común entre casi todas las aplicaciones web, debido a eso, este problema es cubierto usualmente por un proveedor de autenticación.

  El protocolo *OpenID Connect* es el seleccionado para solucionar este problema, la razón principal es que los flujos propuestos son ideales para autenticar hacia varios servicios, lo cual con esta tecnología se puede hacer con facilidad.

  Además es compatible con el patrón de seguridad de microservicios de uso de *access token* para autenticar hacia los servicios desplegados para la aplicación Aliadas [Richardson, 2019c].

  Las soluciones provistas por la lista de servicios certificados por la fundación *OpenID* (ubicada en la sección de anexos) es una buena referencia de qué aplicaciones servirían para implementar este protocolo en nuestro conjunto de servicios desplegados, sin embargo la mayoría de estas soluciones tienden a ser pagadas. Es por esto que se decidió utilizar **Keycloak** que es la tecnología desarrollada por el conjunto de soluciones de **Red Hat** cuyo uno de sus sistemas operativos será usado para ejecutar este servicio, además de que ya se tiene experiencia trabajando y conectando varios servicios con este sistema en el *data center* del departamento de informática.

  En la siguiente figura se muestra el flujo con el que funciona el uso del patrón de uso de *access token*.

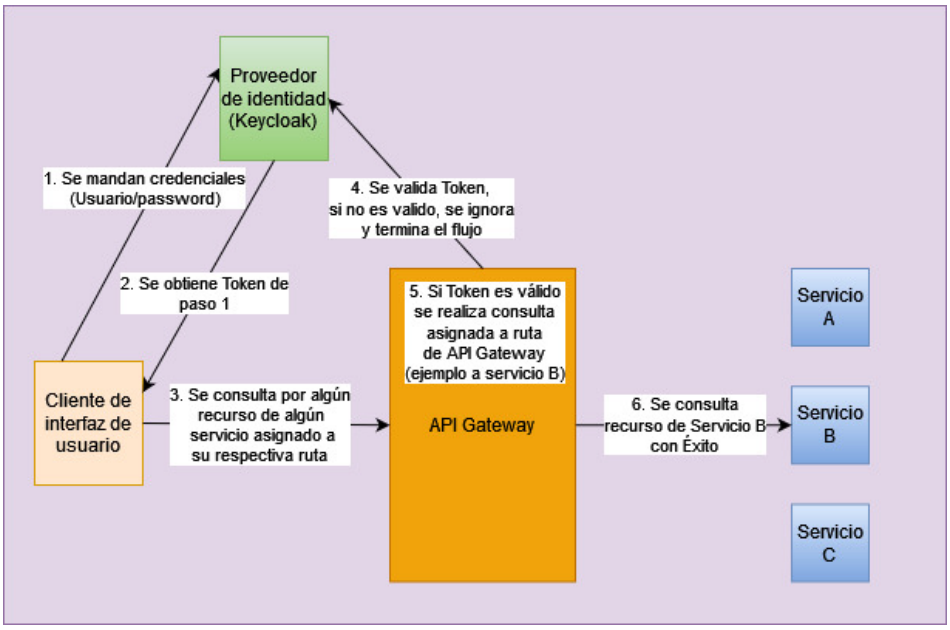


Figura 13: Flujo de inicio de sesión en OpenID Connect y consulta de recurso de algún servicio en arquitectura de software diseñado

Fuente: Elaboración propia.

Un *token* es una representación de información obtenida en este caso del proveedor de identidad, como en este caso, en particular el *token* tiene la información del usuario y los roles de este dentro de la aplicación con los cuales se pueden filtrar recursos de los servicios desplegados. Por ejemplo, un usuario comprador no podría crear productos ni una tienda como podría hacer un usuario vendedor en el contexto de esta aplicación, un token tiene un tiempo de validez, en este proyecto es de 10800 segundos, aparte se usa el estándar de *JSON Web Token* (JWT) para más información de JWTs revisar en anexos.


```
{
    "access_token": "eyJhbGciOiJSUzI1N...",
    "expires_in": 10800,
    "refresh_expires_in": 0,
    "refresh_token": "eyJhbGciOiJIUzI1N...",
    "token_type": "Bearer",
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5c...",
    "not-before-policy": 0,
    "session_state": "6286c033-...",
    "scope": "openid phone profile rut address avatar offline_access email website gender"
}
```


Figura 14: Formato de datos entregados al iniciar sesión, algunos datos fueron truncados por poder poseer información sensible del sistema

Fuente: Respuesta de servicio de proveedor de identidad **Keycloak**.

En el paso 1 se mandan las credenciales para iniciar sesión desde el cliente de interfaz

de usuario, en este caso, una instancia de la aplicación móvil de Aliadas al servicio de proveedor de identidad.

Figura 15: Vista de inicio de sesión
Fuente: Aplicación Móvil Aliadas.

En el paso 2 se obtiene el token desde el proveedor de identidad con los datos equivalentes a los datos del usuario que inició sesión en el sistema, de aquí en adelante siempre se debe de mandar el *token* en el encabezado de las consultas para poder obtener datos de los servicios a través del *API Gateway*, sino, se rechazarán las solicitudes.

Posteriormente en el paso 3 y una vez se inicia sesión, se adjunta el token en el encabezado de la consulta y se manda a la ruta asignada al servicio al que se realiza la consulta en el *API Gateway*, luego en el paso 4 se valida el token, si no es válido, el flujo termina ahí, debido a que el usuario deja de estar autenticado, si esta autenticado, se considera efectiva la consulta en el paso 5 para luego llegar directamente al servicio a hacer la consulta necesaria para que la aplicación muestre los datos adecuadamente en el paso 6.

En la implementación realizada en estricto rigor la validación del token se encuentra en cada servicio a través del uso de un middleware que es una función que se ejecuta antes de aceptar la consulta mandada al servicio cuya lógica esta en este mismo que viene por defecto en una plantilla del *framework* que se realizó en el proyecto Aliadas, sin embargo el proceso es el mismo solo que sucede en el servicio al que la consulta se dirige. Esto se realizó debido a la implementación del usuario invitado dentro del sistema, cuya funcionalidad está en el servicio de manejador de API de administrador de proveedor de identidad.

Keycloak además, presenta hartos adaptadores por defecto para realizar *single sign on*, es decir, utilizar un proveedor externo de autenticación para obtener datos del usuario y autenticar dentro de el conjunto de servicios del proyecto Aliadas, ofreciendo facilidades para desarrollar nuevos adaptadores para plataformas que usen **Oauth 2** o **OpenID Connect**.

Se utilizaron los adaptadores de autenticación para *single sign on* para autenticarse a través de *Google* cuyo adaptador venía por defecto y *Apple* en el que se tuvo que utilizar un adaptador externo ya que no estaba por defecto en la lista de adaptadores, sin embargo, se tuvo que desarrollar un adaptador debido a que no existía, para autenticar utilizando el proveedor de identidad de **Mercado Libre**, el cual fue desarrollado e implementado sin problemas, la documentación y repositorios de los adaptadores se encuentra en anexos.

Esto cambia un poco el flujo de inicio de sesión mencionado anteriormente como se muestra en la siguiente figura.

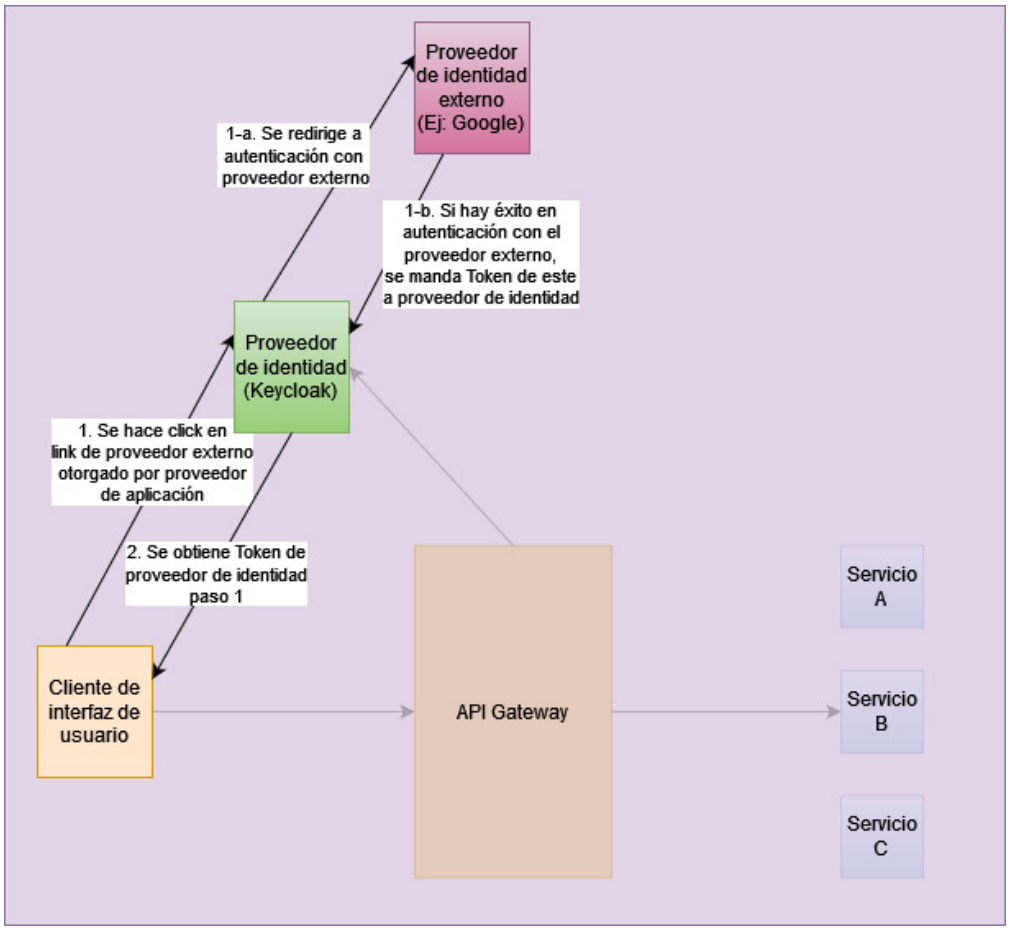


Figura 16: Muestra de flujo de inicio de sesión a través de uso de proveedor de identidad como intermediario (*broker*) de proveedores de identidad externos
Fuente: Elaboración propia.

En el paso 1 se hace “click” en el enlace de proveedor externo otorgado por proveedor de aplicación, como se muestra en la figura 15 en la sección de “También puedes ingresar con”, posteriormente en el paso 1-a, se redirige a la vista de autenticación con el proveedor externo, se autentica y si se tiene éxito, se manda el *token* obtenido del proveedor externo en el paso 1-b, posteriormente, se guarda el *token* del proveedor externo en el proveedor de identidad de la propia aplicación para poder iniciar sesión a través de este proveedor externo a futuro.

La utilidad de guardar el token del proveedor externo es que se pueden acceder a recursos de este proveedor externo, el uso que se da a esto es a la consulta de productos que por ejemplo, tiene un usuario vendedor en *marketplaces* como **Mercado Libre**.

En este flujo se utiliza el proveedor de identidad **Keycloak** como intermediario (*broker*) entre el cliente a autenticar y el proveedor de identidad externo, en anexos se puede revisar la documentación correspondiente.

- *Backing Services*:

La selección del conjunto de soluciones de tecnologías de *backing services* se puede dividir en 2 tipos:

- Persistencia de datos:

  Debido a que se levantarán varios servicios con sus respectivas funcionalidades y modelos de persistencia, se puede utilizar por ejemplo una tecnología distinta por cada servicio para persistir los datos manejados por estos, no obstante no es la idea, pues mientras más tecnologías o soluciones se despliegan, más de estos se deberán mantener, lo cual añade complejidad a la mantención del conjunto de servicios de la aplicación.

  Aún así, cada servicio tiene sus propias necesidades de persistencia por lo que en este proyecto se usaron principalmente 4 tecnologías de persistencia según las necesidades de la aplicación; **Postgresql**, **MongoDB**, **Elasticsearch** y **Amazon S3**.

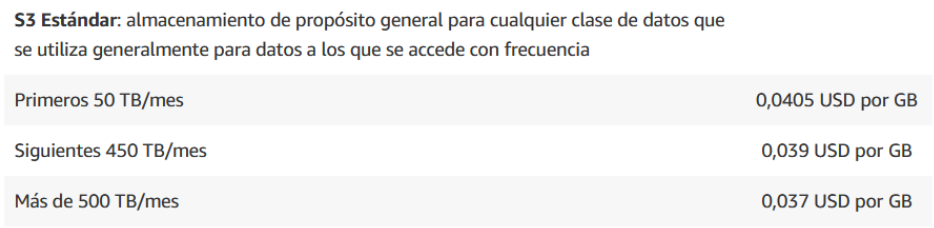
  Postgresql es un motor de base de datos relacional que se utilizó en los servicios de *e-commerce* y de proveedor de identidad, se utilizó este *backing service* debido a que la documentación de las tecnologías de **Keycloak** y **Spree** respectivamente recomendaban utilizar esta tecnología para persistir los datos de estos servicios, esto aparece en la documentación de **Keycloak** y **Spree** adjunta en la sección de anexos.

  MongoDB es un motor de base de datos basada en documentos que se utilizó para el servicio de registro de acciones y el servicio de chat. Se utilizó para el servicio de registro de acciones debido a lo maleable que es el esquema para probar si el modelo definido en la aplicación a través de la herramienta de mapeo de objeto a documento corresponde con lo que se necesita para guardar acciones y posteriormente ser consultado por el servicio del sistema recomendador. Por otro lado se utilizó en el servicio de chat debido a que la documentación de la tecnología **Rocket Chat** utilizaba esto como *backing service*, esto sale mencionado en su documentación adjunto en la sección de anexos. Además es importante señalar que es mucho mas sencillo de mantener y escalar comparado a una base de datos relacional pues permite con facilidad utilizar conjuntos de replicas para respaldo de los datos o *sharding* en caso de tener bastantes instancias para dividir los datos entre las instancias de **MongoDB** levantadas.

  Elasticsearch es un motor de analítica y búsqueda para aplicaciones, entre sus muchas funcionalidades se utilizó principalmente la de indexación de productos, vendedores y ofertas del servicio de *e-commerce* para no sobrecargar la base de datos relacional levantada con **Postgresql** debido a que la cantidad de productos y vendedores a largo plazo crecerá de forma indefinida. Existen otras opciones como por ejemplo **Apache Solr** o **Google Cloud Search**, pero se decidió utilizar **Elasticsearch** debido a su fácil integración con **Ruby on Rails** y **Spree** además de que al ser *open source*, no se tiene problemas para desplegar y mantener este *backing service*

  Amazon S3 es un servicio *cloud* de almacenamiento que se utilizó particular-

mente para guardar las imágenes de productos y vendedores del servicio de *e-commerce*. Existen otras opciones como por ejemplo **MinIO** o **Paperclip**, sin embargo resulta ser bastante conveniente para el precio el usar **Amazon S3**, para guardar las imágenes debido a que a futuro no habrá que preocuparse por pérdida de datos por corrupción de datos del disco donde estén guardadas las imágenes. Además mantener servicios de almacenamiento, es particularmente caro y consume bastante hardware si es que se quiere hacer correctamente y evitando problemas de corrupción de datos. En la siguiente figura se muestran los precios de almacenamiento en el servidor de São Paulo a la fecha de emisión de este trabajo.

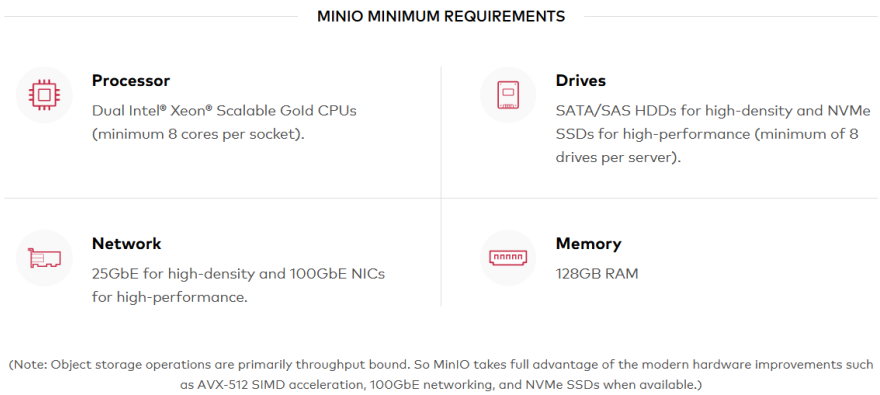


S3 Estándar: almacenamiento de propósito general para cualquier clase de datos que se utiliza generalmente para datos a los que se accede con frecuencia

Primeros 50 TB/mes	0,0405 USD por GB
Siguientes 450 TB/mes	0,039 USD por GB
Más de 500 TB/mes	0,037 USD por GB

Figura 17: Precios de almacenamiento de servicio *Amazon S3* en servidor de *São Paulo*, *Brasil*
Fuente: Página web de *Amazon Web Services*
`https://aws.amazon.com/es/s3/pricing/`

Además se muestra en la siguiente figura los requisitos mínimos para correr el servicio de **MinIO** en producción en arquitecturas de infraestructura *on premise*, notar que los requisitos son bastante caros para ser desplegados de esta forma en producción y asumiendo que a largo plazo se puede efectivamente tener que utilizar esta capacidad de *hardware*.



MINIO MINIMUM REQUIREMENTS

Processor
Dual Intel® Xeon® Scalable Gold CPUs (minimum 8 cores per socket).

Drives
SATA/SAS HDDs for high-density and NVMe SSDs for high-performance (minimum of 8 drives per server).

Network
25GbE for high-density and 100GbE NICs for high-performance.

Memory
128GB RAM

(Note: Object storage operations are primarily throughput bound. So MinIO takes full advantage of the modern hardware improvements such as AVX-512 SIMD acceleration, 100GbE networking, and NVMe SSDs when available.)

Figura 18: Especificaciones mínimas de hardware para desplegar el *backing service* **MinIO** en infraestructuras *on premise*
Fuente: Página web de **MinIO** `https://min.io/product/reference-hardware`

- Consistencia de datos entre servicios:

  La labor de este *backing service* es permitir asegurar consistencia de datos entre servicios, debido a que ciertas transacciones pueden requerir algún cambio que se tenga que hacer en otro servicio, por ejemplo, si se tiene un servicio de productos y otro de tiendas, si se borra una tienda, lo más probable es que se tengan que borrar sus respectivos productos, por lo tanto si o si hay que realizar una operación de borrado en la base de datos en ambos servicios.

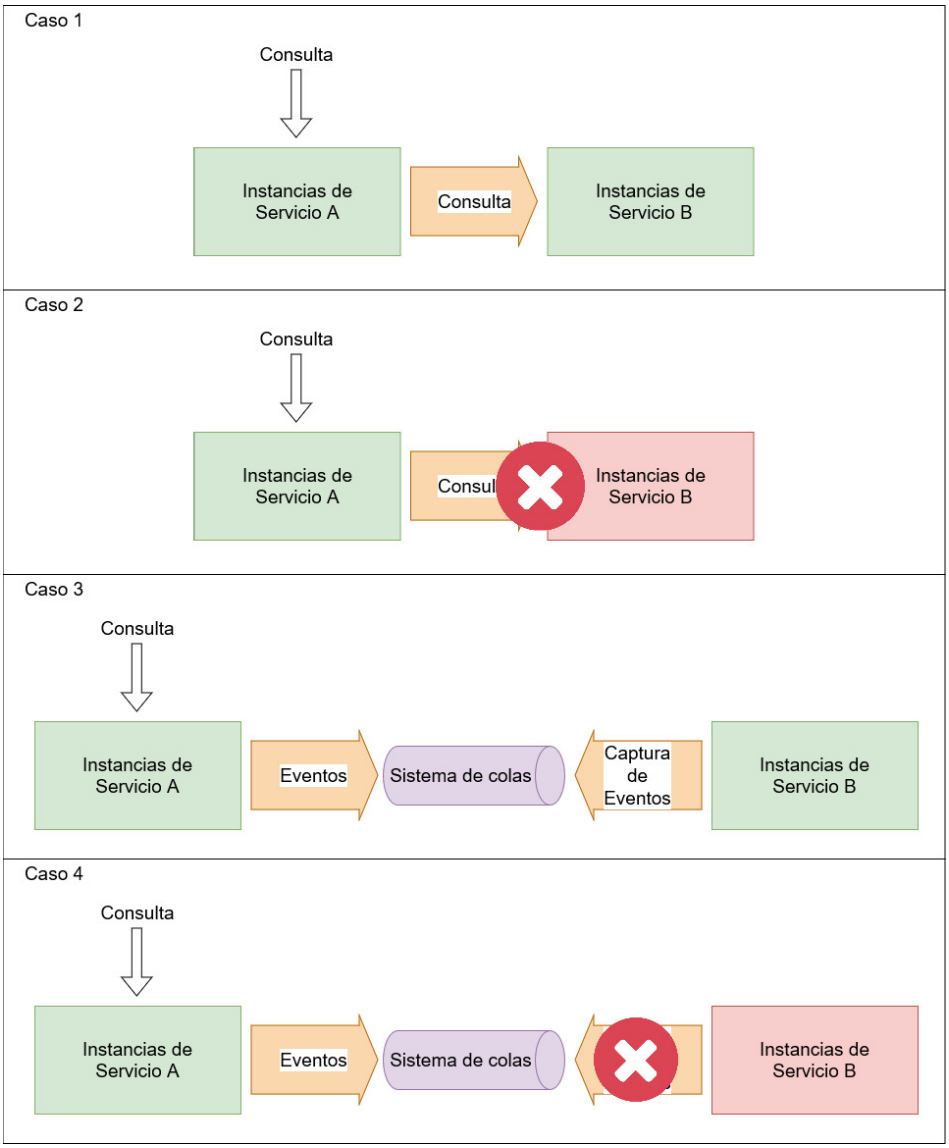


Figura 19: Ejemplos de uso y desuso de *backing service* de consistencia de datos
Fuente: Elaboración propia.

Ahora, supongamos que no se implementa un *backing service* para tener consistencia de datos entre servicios, en ese caso el curso intuitivo al desarrollar una

aplicación es hacer una consulta del servicio A al servicio B con un cliente *HTTP*, en la figura siguiente se ilustran distintos casos, siendo A y B casos en los que se aplica este ejemplo en que no se utiliza un *backing service* para mantener consistencia de datos entre servicios. Si se aplica esta forma, entonces se asume que siempre los servicios A y B estarán arriba como se muestra en el caso 1, sin embargo, asumir que los servicios A y B estarán arriba en todo momento es un caso extremadamente ideal por lo que si llega una consulta al servicio A y el servicio B está abajo, puede ocurrir que se cree una inconsistencia entre el servicio A y B como se muestra en el caso 2 pues, siguiendo el ejemplo de productos y tiendas, si se borra una tienda en A y no se puede acceder al servicio B que maneja los productos, entonces la aplicación quedará con productos sin vendedor dando a lugar una inconsistencia entre datos de los servicios.

Por otro lado, si se usa un sistema de colas, como el caso 3, no se mandan consultas al otro servicio, sino que se mandan eventos con los datos necesarios mandados desde el servicio A para el otro servicio B que se guardan en el sistema de colas que sería nuestro *backing service* y serían posteriormente consuimdos por instancias del servicio B. Lo anterior cubre el caso mostrado en el caso 4 en donde en caso de que el servicio B este abajo, el *backing service* se encarga de mantener los datos mandados hasta que alguna instancia del servicio B se encargue de consumir el dato o los datos en cola, mandados desde algunas de las instancias del servicio A.

Soluciones para implementar esto, hay varias, entre ellas **Redis**, **ZeroMQ**, **Apache Kafka**, entre otros. Sin embargo, se decidió desplegar **RabbitMQ**, por su facilidad de uso, escalabilidad y por ofrecer adaptadores para distintos lenguajes de programación, por ejemplo en este proyecto en el que en los servicios desarrollados predominan los lenguajes **Ruby** y **Python** se utilizaron los paquetes **bunny** y **pika** respectivamente para la conexión hacia este *backing service*, más detalles sobre estas librerias y la tecnología se pueden encontrar en el enlace de su documentación en la sección de anexos.

- Despliegue de *backing services*:

  El despliegue de *backing services* se hizo de manera normal, es decir sin contenedores, sino que en máquinas virtuales, esto para tener mejor ordenado los mensajes de estos al buscar algún error, además por experiencia personal, si se utilizaba contenedores, la corrupción de datos era más recurrente cuando se desplegaba en contenedores.

  Otra opción es usar tecnologías *cloud* para esto, pero los precios para desplegar persistencia eran demasiado caros para usar esos despliegues, especialmente el precio de instancias de **PostgreSQL** y **MongoDB**

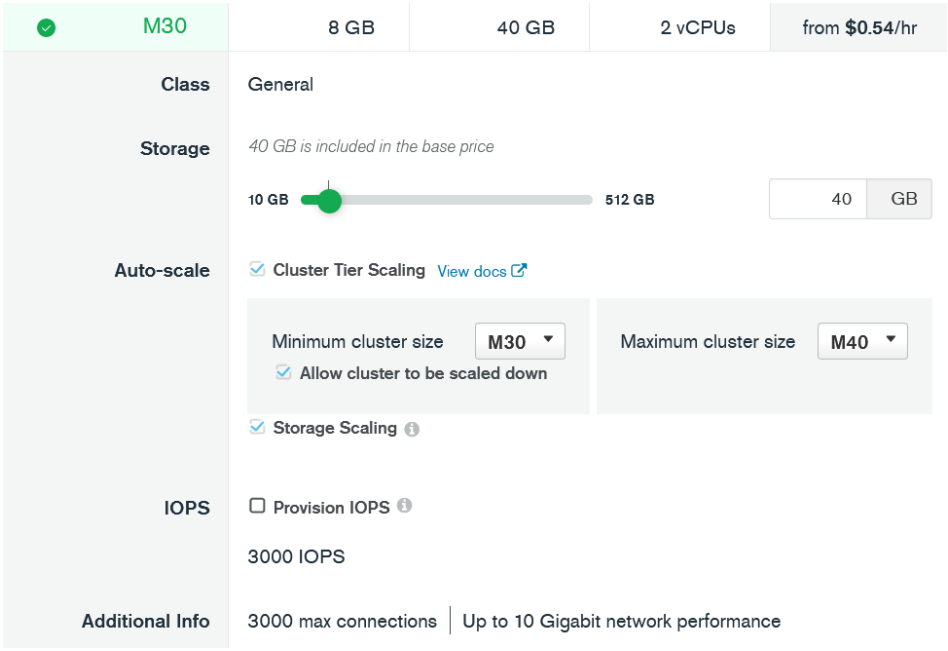


Figura 20: Precio de instancia ideal para proyecto Aliadas en **Mongo Cloud**
Fuente: Calculadora de precios de Mongo Cloud `https://cloud.mongodb.com/`.

Como se muestra en la figura 20, se cobra por hora, por lo que en un mes se cobraría $388 dolares o alrededor de $318.000 pesos chilenos, lo cual sería posible pedir si se usa el dinero netamente para **MongoDB** considerando que se tiene $2.600.000 de pesos chilenos para tecnologías *cloud*.

Por otro lado como se menciona anteriormente, tal vez el único *backing service cloud* que valga absolutamente la pena usar es *Amazon S3* por lo barato que ese el almacenamiento.

- Sistema operativo de máquinas virtuales:

  Se utilizó principalmente el sistema operativo *CentOS 7*, en estas máquinas es donde se inician los contenedores respectivos de los servicios desplegados principalmente o servicios desplegados normalmente en algunos casos.

  Se utiliza este sistema operativo porque es recomendado para su uso en servidores y por la experiencia que se tiene utilizando este sistema operativo y sus opciones de seguridad como por ejemplo **SELinux** para el despliegue de servicios u aplicaciones.

- Herramientas de despliegue de servicios:

  Considerando que se usará el patrón de despliegue de uso de contenedores en un *host*, se utilizará **docker** y **docker compose** para desplegar los servicios en el servidor que tendrá los servicios desplegados.

  Existen opciones más completas y complejas a la vez que permiten escalar automáticamente las aplicaciones desplegadas como por ejemplo **Kubernetes** sin embargo se

decidió no utilizar esta tecnología inicialmente debido al perfil de las personas que tenderán a mantener a futuro esta aplicación.

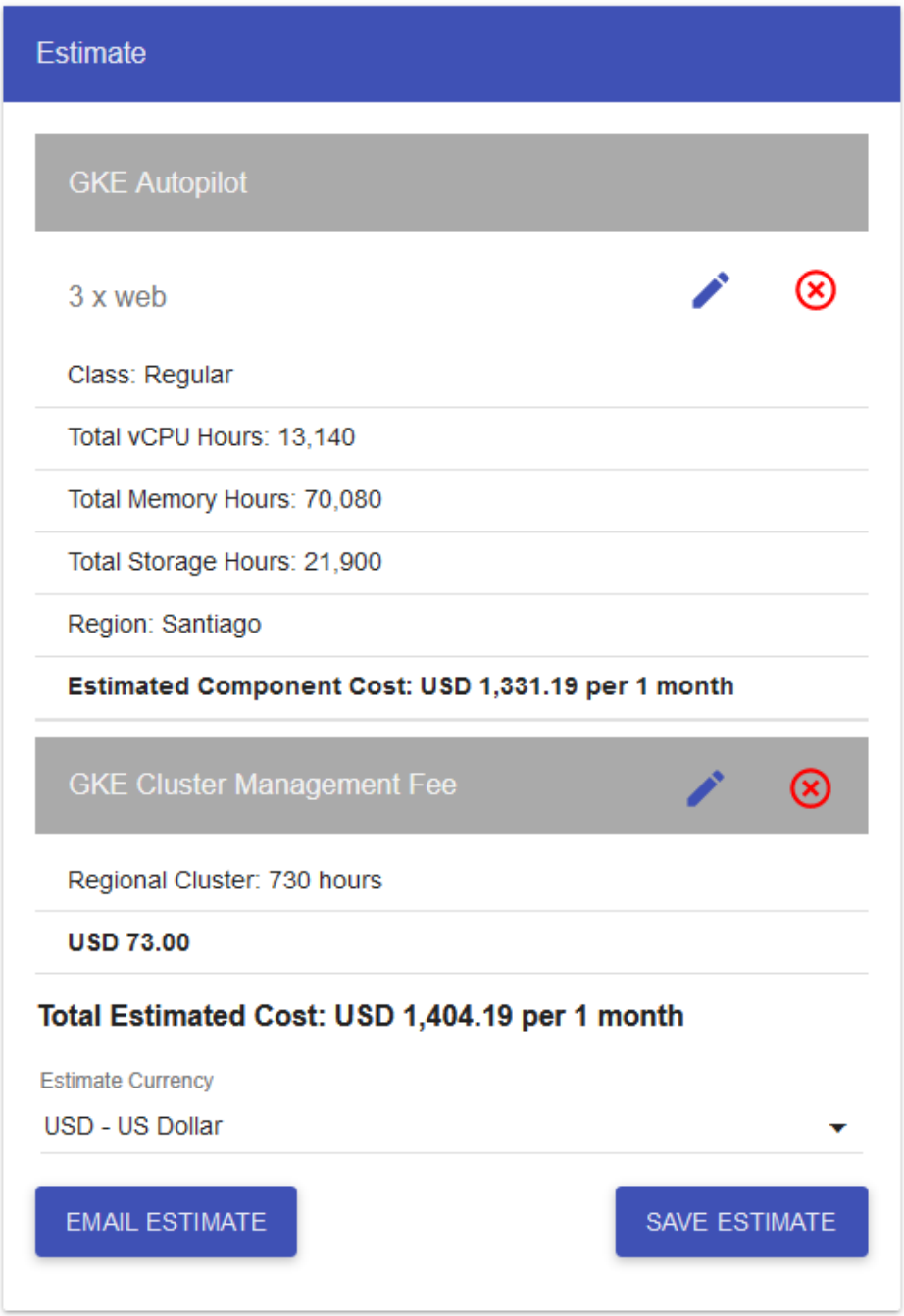


Figura 21: Cotización hecha sobre **Google Kubernetes Engine**
Fuente: Calculadora de precios de **GKE** `https://cloud.google.com/products/calculator/#id=2da17bfc-daa5-47ef-9522-c6dc69fa3179`.

Por otro lado existen opciones en la nube que abstraen la implementación de un or-

questador como **Kubernetes** en por ejemplo una arquitectura de infraestructura *on premises* como por ejemplo **Amazon Elastic Container Service (Amazon ECS)**, **Google Kubernetes Engine (GKE)**, **IBM Cloud Kubernetes Service**, entre otros. Sin embargo sus precios son altísimos para desplegar los contenedores de los servicios que estaríamos desplegando para la aplicación Aliadas pensando que es un proyecto de alrededor de un año y aún sin ingresos, la siguiente figura muestra una cotización hecha sobre **GKE** simulando el hardware mínimo que se tiene desde el servidor de despliegue de servicios de la aplicación en el datacenter (mencionado en sección Hardware utilizado).

Es por esto que se decidió utilizar como infraestructura inicial la propuesta en el presente trabajo con los contenedores desplegados con **docker compose** además de que se pueden pedir máquinas virtuales a través de tickets para la organización del proyecto de Aliadas al departamento de informática de la universidad.

- Configuración de certificados SSL:

  El protocolo SSL o *Secure Sockets Layer* permite encriptar, es decir, ocultar los datos enviados desde un computador mediante una clave las consultas *HTTP* que en principio van en texto plano por internet, estos se realiza a través de certificados en los servidores que corren los servicios que contienen la llave para desencriptar las consultas *HTTP*.

  Para gestionar y actualizar estos certificados se utiliza **certbot** que es una herramienta *open source* para automatizar el uso de certificados SSL de **Let's Encrypt** que es una autoridad certificada en creación de certificados SSL.

  Al día de hoy todos los servicios tienen certificado TLS v1.3 que es la más actualizada al día de hoy.

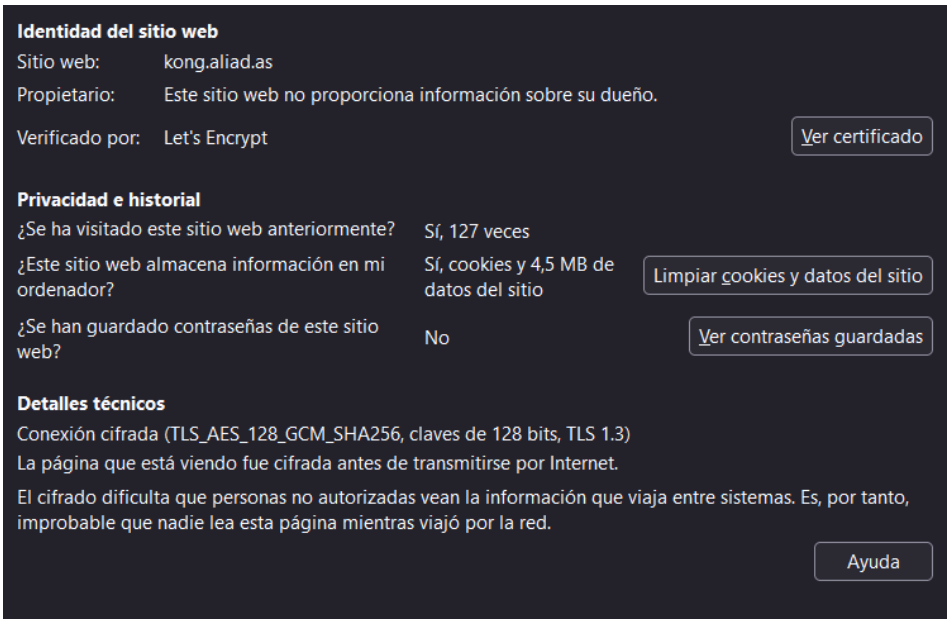


Figura 22: Información de certificado SSL de despliegue de *API Gateway*
Fuente: Despliegue de *API Gateway*.

CAPÍTULO 4
EVALUACIÓN DEL DESPLIEGUE DE ARQUITECTURAS DE SOFTWARE E INFRAESTRUCTURA

4.1. Arquitectura de Software

4.1.1. Prácticas de desarrollo

En la presente sección se mostrarán ejemplos de practicas utilizadas en servicios desplegados para la aplicación basada en los 12 factores mencionados en la sección “Marco conceptual”.

- Código Base:

  Las aplicaciones desplegadas en los servidores cumplen con estar dentro de un repositorio el cual guardaría el código base de cada servicio.

  En figura 23 se aprecian algunos ejemplos de esto. Cada servicio tiene sus respectivos despliegues teniendo como mínimo el despliegue de producción.

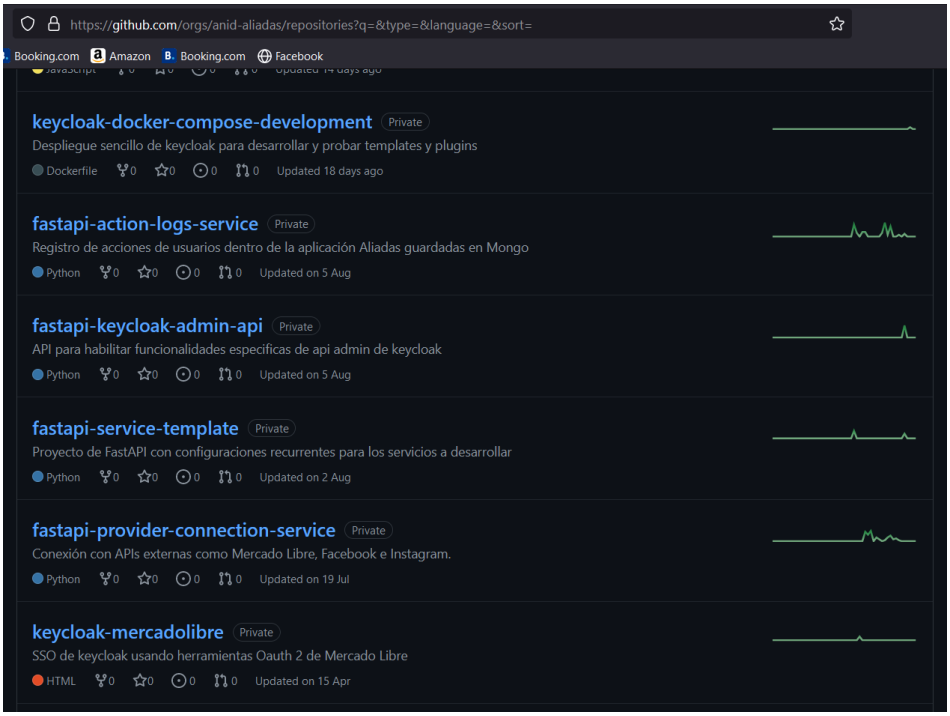


Figura 23: Algunos códigos bases de los servicios desplegados para la aplicación Aliadas. Fuente: Repositorio de organización anid-aliadas en *Github*.

Por ejemplo, en este momento se esta corriendo 2 despliegues, uno de producción y otro de prueba para el servicio de *e-commerce* como se muestra en la figura 24.

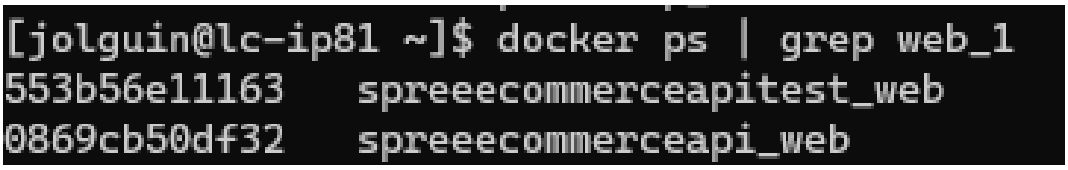


Figura 24: Despliegues de servicios de *e-commerce* en contenedores, de prueba y de producción respectivamente
Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Dependencias:

  Cada servicio tiene sus dependencias respectivas. Por ejemplo en el servicio de registro de acciones al estar desarrollado en *Python* se utiliza *pip* para definir las dependencias como se muestra en la figura 25.

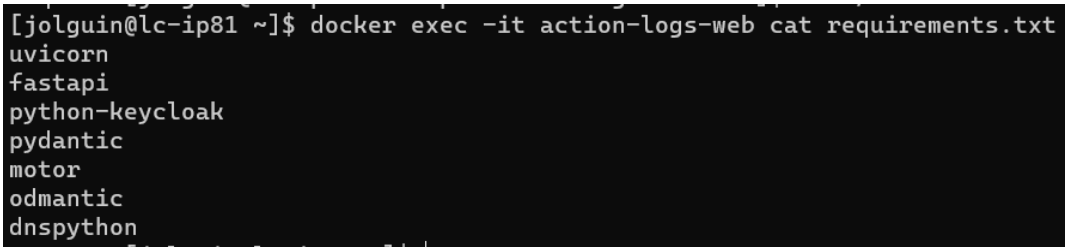


Figura 25: Dependencias de servicio de registro de acciones en producción
Fuente: Consola de servidor de despliegue de servicios de la aplicación.

La rutina para instalar las dependencias está definida en el *Dockerfile* del proyecto por lo que se hace cada vez que se hace un despliegue de la aplicación al crear el contenedor 26.

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8

# Copiar archivos a imagen de docker
COPY requirements.txt /app/requirements.txt
COPY ./.env /app/.env

# Archivos de paquete "app"

COPY ./app /app/app

# Instalar dependencias

RUN pip install -r requirements.txt
```

Figura 26: *Dockerfile* de servicio de registro de acciones en producción
Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Configuraciones:

  Cada servicio tiene sus configuraciones respectivas, la forma en que se trabajó esto es que cada servicio tiene en el repositorio un archivo de variables de entorno de ejemplo como el de la figura 27, esto para mostrar las variables que necesita el servicio para poder ser desplegado.

  Notar que cada despliegue tiene su propio conjunto de valores de variables de entorno propios, estos varían según la dirección de los *backing services*, los cuales son usados según lo desarrollado en el servicio.

```
[jolguin@lc-ip81 spree-ecommerce-api]$ cat .env.example
POSTGRES_HOST=db
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
POSTGRES_DB=spree_rails
PGUSER=postgres
PGPASSWORD=postgres
ELASTIC_HOST=elasticsearch
ELASTIC_USER=elastic
ELASTIC_PASSWORD=changeme
KEYCLOAK_SERVER_URL=https://keycloak.aliad.as/auth/
KEYCLOAK_REALM_ID=aliadas
KEYCLOAK_CLIENT_ID=
KEYCLOAK_CLIENT_SECRET=
RAILS_ENV=development
SECRET_KEY_BASE=

S3_ENDPOINT=http://0.0.0.0:9000
S3_BUCKET_NAME=aliadas-test
S3_REGION_NAME=
S3_ON_PREMISE=
S3_ACCESS_KEY_ID=minioadmin
S3_SECRET_ACCESS_KEY=changeme
```

Figura 27: Archivo de ejemplo de variables de entorno de servicio *e-commerce*
Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- *Backing services*:

  Las tecnologías de *backing services* utilizados por los servicios desplegados son:

  - PostgreSQL: Motor de base de datos relacional, utilizado principalmente por el servicio de *e-commerce* y el servicio de proveedor de identidad.
  - MongoDB: Base de datos basado en documentos, utilizado principalmente por el servicio de guardado de acciones de usuario.
  - Elasticsearch: Motor de analítica utilizado para apoyar consultas de búsqueda de bases de datos, este es utilizado principalmente por el servicio *e-commerce* y el sistema recomendador.
  - Postfix: Servidor *SMTP* para mandar correos electrónicos, es utilizado principalmente por el servicio de proveedor de identidad.
  - Amazon S3: Servicio de almacenamiento de objetos en la nube. Se utiliza para guardar las imágenes en producción, este servicio se añadió para desacoplar el almacenamiento de imágenes del sistema de archivos de la aplicación misma.

- RabbitMQ: Servicio de manejo de colas, es utilizado para manejar la consistencia entre servicios desplegados.

En particular en el servicio *e-commerce* se utilizan PostgreSQL, Elasticsearch y Amazon S3

en la figura anterior se muestra como se ingresan las conexiones y credenciales respectivas para utilizar los *backing services* a través de variables de entorno, práctica que se repite en el resto de los servicios desplegados.

- Construir, desplegar, ejecutar:

  Los servicios desarrollados tienen sus respectivos etiquetados según las versiones que van teniendo según los requisitos funcionales que van siendo desarrollados o modificados durante el proyecto.

  En la figura 28 se muestra el etiquetado de las versiones desarrolladas del servicio de registro de acciones cada uno con sus respectivos cambios asociados al código base.

Figura 28: Etiquetado de distintas versiones lanzadas para el servicio de registro de acciones de usuario

Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Procesos:

  Las aplicaciones desplegadas no manejan estados de por si, o en otras palabras, no guardan datos que se puedan perder en caso de caída de la instancia del servicio al ser por definición "*stateless*", sino que persisten estos estados usando los *backing services* definidos anteriormente.

- Asignación de puertos:

  Los servicios y sus respectivas instancias quedan asignadas a sus respectivos puertos dentro de la infraestructura.

  Como se observa en el siguiente ejemplo en la figura 29 donde se muestran 2 servicios de los que están en funcionamiento en la aplicación dentro de sus respectivos contenedores, cada servicio tiene su respectivo puerto asociado para permitir la comunicación entre estos o para poder exponerse al público en caso de ser necesario.

Figura 29: Al consultar los contenedores que se están corriendo se pueden consultar los puertos que se están utilizando, aquí se muestran 2 servicios de esta lista, el servicio de registro de acciones de usuario y el servicio de chat

Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Concurrencia:

  Las aplicaciones desarrolladas dentro de este proyecto están diseñadas para que estos puedan escalar horizontalmente con facilidad, es decir, correr más de una instancia de la aplicación dentro de la arquitectura de infraestructura sin otorgar problemas de consistencia de los datos persistidos.

  En la figura 30 se muestra un ejemplo de 3 instancias del servicio de registro de acciones de usuario a los cuales se puede balancear carga por ejemplo con un servidor web o con el mismo servicio de *API Gateway*

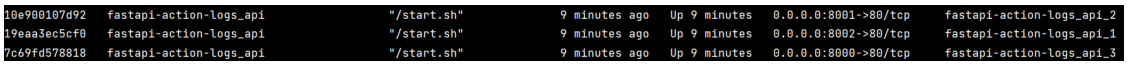


Figura 30: Aquí se aprecia un despliegue de 3 instancias del servicio de registro de acciones de usuario

Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Desechabilidad:

  Las aplicaciones al no persistir estado por su cuenta (sino que a través de "*backing services*") estos tienen menos posibilidad de dar a lugar problemas de consistencia de datos y corrupción de estos en caso de caídas de los servicios.

- Igualdad en desarrollo y producción:

  La única diferencia entre los despliegues de desarrollo y producción son en los valores de las configuraciones entre ellos, estos se mostrarían pero los valores de configuración son credenciales de carácter sensible.

- Historiales:

Los historiales de la aplicación se pueden gestionar sin problemas dentro de los ambientes de los contenedores de cada servicio, en la figura 31 se muestra un ejemplo del servicio de registro de acciones de usuario, donde en caso de ser necesario se pueden observar los eventos que ocurren asociados al servicio ya sean consultas o fallas que este pueda llegar a tener.

```
172.17.0.1:37624 - "GET /logs_query/products?page=0&per_page=1000 HTTP/1.1" 200
172.17.0.1:37624 - "GET /logs_query/products?page=1&per_page=1000 HTTP/1.1" 200
172.17.0.1:37746 - "GET /logs_query/vendors?page=0&per_page=1000 HTTP/1.1" 200
172.17.0.1:37754 - "GET /logs_query/vendors?page=1&per_page=1000 HTTP/1.1" 200
172.17.0.1:37844 - "GET /logs_query/vendors?page=0&per_page=1000 HTTP/1.1" 200
172.17.0.1:37852 - "GET /logs_query/vendors?page=1&per_page=1000 HTTP/1.1" 200
172.17.0.1:37852 - "GET /logs_query/products?page=0&per_page=1000 HTTP/1.1" 200
172.17.0.1:37852 - "GET /logs_query/products?page=1&per_page=1000 HTTP/1.1" 200
```

Figura 31: Aquí se muestra parte del historial de una instancia del servicio de registro de acciones de usuario

Fuente: Consola de servidor de despliegue de servicios de la aplicación.

- Administración de procesos:

  Los servicios tienen sus propias herramientas para realizar las mantenciones necesarias, en el presente ejemplo de la figura 32 se muestra la rutina para crear los índices del backing service **Elasticsearch** nuevamente desde el servicio de *e-commerce* para actualizar la búsqueda de productos.

  A la vez se muestra la actualización de productos del sistema recomendador en el conjunto de rutinas siguiente.

```
#!/bin/sh

# Elastic indexation
docker exec -t spreeecommerceapi_web_1 rake elastic:reset_indexes[force]

# Recommender system update
docker exec -t recommender-system-web python ../update_products_script.py &&
docker exec -t recommender-system-web python ../update_recommender_script.py
&& docker restart recommender-system-web
```

Figura 32: Aquí se muestra un rutina que utiliza parte de las herramientas de administración de procesos

Fuente: Consola de servidor de despliegue de servicios de la aplicación.

4.2. Arquitectura de Infraestructura

Para la validar la arquitectura de infraestructura se realizaran pruebas de estrés a las consultas de servicios críticos, en particular a las consultas de búsqueda, recomendación y guardado de acciones, se harán test de estrés a servicios críticos de la aplicación usando el software

Apache Jmeter que es una herramienta para probar las consultas por segundo que puede soportar una *API* desplegada.

Recordar que los servicios desplegados están configurados para tener un proceso por núcleo del servidor donde está desplegado, es decir, tienen 6 instancias corriendo, se decidió sin embargo cambiar configuraciones del servicio de *e-commerce* para probar los distintos resultados de las pruebas de estrés.

4.2.1. Pruebas de estrés de los servicios desplegados

Para las pruebas de estrés se consideró uno de los flujos más usuales utilizados dentro de la aplicación, el cual se explicará a continuación.

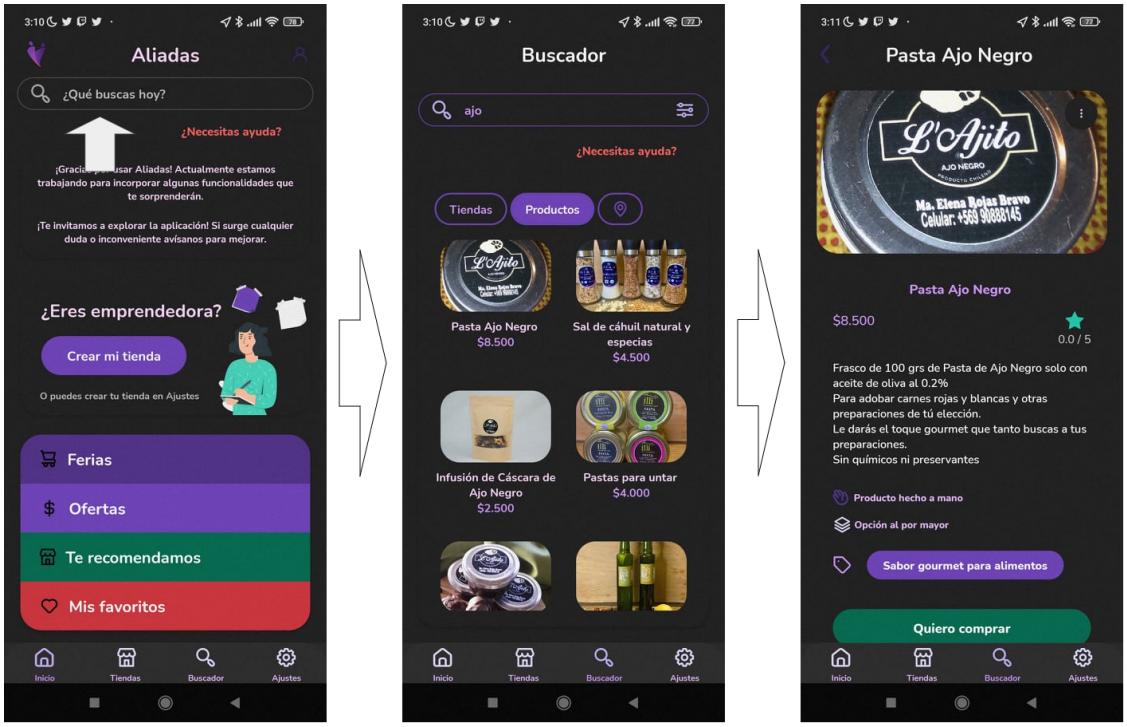


Figura 33: Demostración de flujo que se simulará en prueba de estrés
Fuente: Aplicación Móvil Aliadas.

en la figura 33 se muestra el flujo que se simulará en las pruebas de estrés, en la primera imágen se aprecia la vista incial, en donde se va a ingresar un texto para buscar sobre algún producto, en este caso usamos la palabra “ajo”, luego en la segunda imágen, la vista nos muestra los resultados de esa búsqueda, posteriormente, se selecciona uno de los productos resultantes de la búsqueda y se ingresa a la vista de el producto en si con mayor detalle que es lo que se muestra en la tercera imágen.

En este flujo participan principalmente 3 consultas, las cuales serán las probadas en la prueba de estrés.

La primera consulta proviene del servicio del sistema recomendador el cual consultando a través de una palabra, en este caso “ajo”, entrega resultados obtenidos a través del servicio de búsqueda, a la vez, si es que el usuario ha revisado productos anteriormente, el sistema recomendador, utilizará el servicio de registro de acciones para recomendar productos según las acciones que ha realizado el usuario anteriormente.

La segunda consulta proviene del servicio de *e-commerce* que es la consulta para obtener los datos específicos del producto seleccionado.

La tercera consulta proviene del servicio de acciones que es utilizado cuando se revisa un producto para en este caso en particular, registrar la acción de “visita” del producto seleccionado.

Notar además que todas estas consultas, deben de validar el *token* adjunto en la cabecera de la consulta con el servicio proveedor de identidad, sino los servicios considerarán que el usuario no ha iniciado sesión y los datos consultados no serán entregados.

Para realizar esta prueba de estrés se utilizó el software **JMeter** que permite configurar cuantos usuarios por segundo simular en funcíon de distintos parámetros.

Los parámetros a definir son:

- El número de usuarios a simular o hilos de proceso para mandar consultas.
- El periodo de subida, que consiste en el tiempo en que se demora incrementando linealmente los hilos para llegar al máximo número de hilos mencionado anteriormente.
- Contador de bucle, osea, cuantas veces se quiere repetir las consultas.
- Duración que durará la prueba.

Los valores de los parámetros para la prueba son de 50 hilos para hacerlo calzar con el objetivo de realizar un despliegue inicial que soporte 50 usuarios, 60 segundos de periodo de subida, bucle indefinido y una duración de la prueba de 200 segundos.

A modo gráfico para explicar de mejor manera como funcionan estos parámetros, se muestra en la siguiente figura, los hilos en función del tiempo que estarán ejecutándose durante la prueba, en donde después de 60 segundos estarán 50 hilos haciendo consultas a los servicios hasta el segundo 200

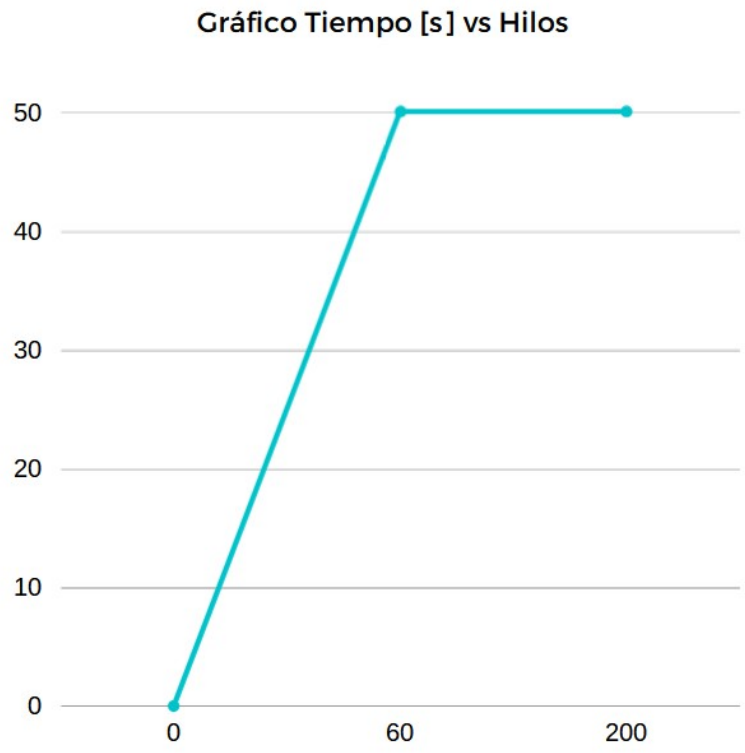


Figura 34: Gráfico de tiempo en segundos ejecutados versus hilos
Fuente: Elaboración propia

Como se mencionó antes, esta prueba simulará el flujo mencionado anteriormente, esto bajo 3 distintas configuraciones.

Primero se mostrará el resultado ejecutando el servicio de *e-commerce* con una sola instancia y las demás con 6 instancias. , también con realizar la prueba con 6 instancias para todos los servicios involucrados y por último, con 6 instancias para todos los servicios involucrados y habilitando el uso de cache en el servicio de *e-commerce* para guardar consultas recurrentes.

Los resultados de la prueba para el caso donde se esta corriendo el servicio *e-commerce* con una sola instancia son los siguientes:

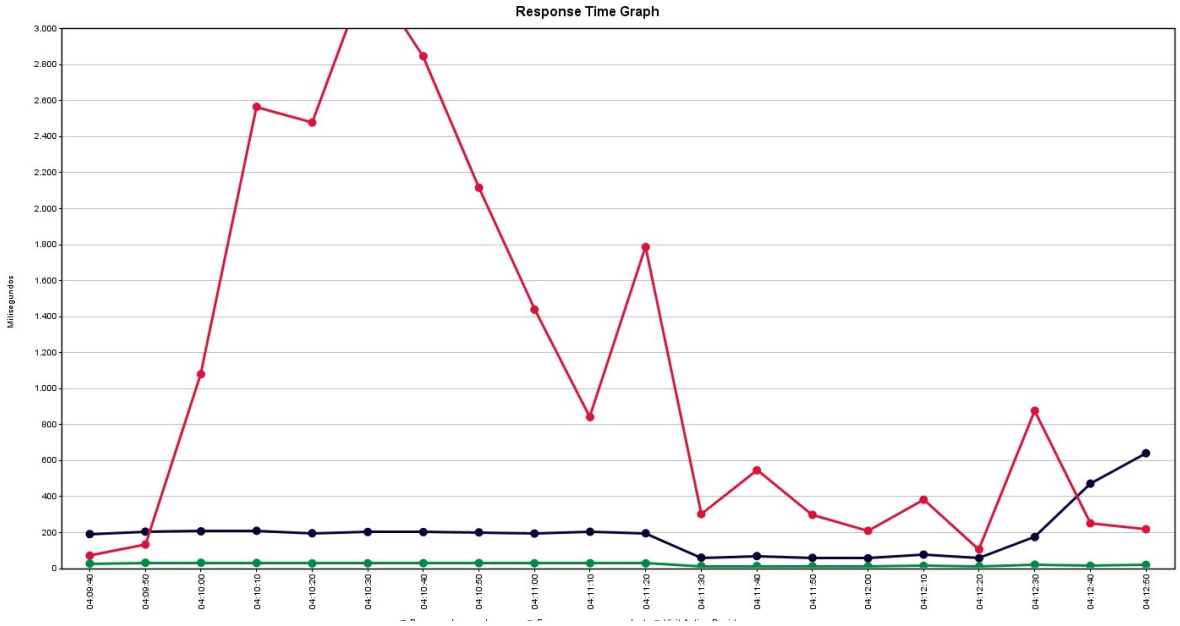


Figura 35: Gráfico de tiempo en segundos ejecutados versus hilos para caso de una sola instancia para servicio *e-commerce*

Fuente: Resultados de JMeter.

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Estándar
Recomender search	11322	144	36	21057	1005,80
E-commerce query ...	11303	603	8	60026	4939,47
Visit Action Register	11273	17	8	83	10,89
Total	33898	255	8	60026	2921,77

Etiqueta	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de Bytes
Recomender search	75,19%	53,2/sec	17,72	82,50	340,8
E-commerce query ...	62,43%	56,1/sec	480,67	86,69	8766,4
Visit Action Register	63,30%	56,5/sec	21,42	93,83	388,5
Total	66,98%	159,4/sec	492,92	252,69	3166,1

Figura 36: Métricas de resultados de caso de una instancia para instancia *e-commerce*

Fuente: Resultados de JMeter.

Los resultados de la prueba para el caso donde se esta corriendo el servicio *e-commerce* con 6 instancias son los siguientes:

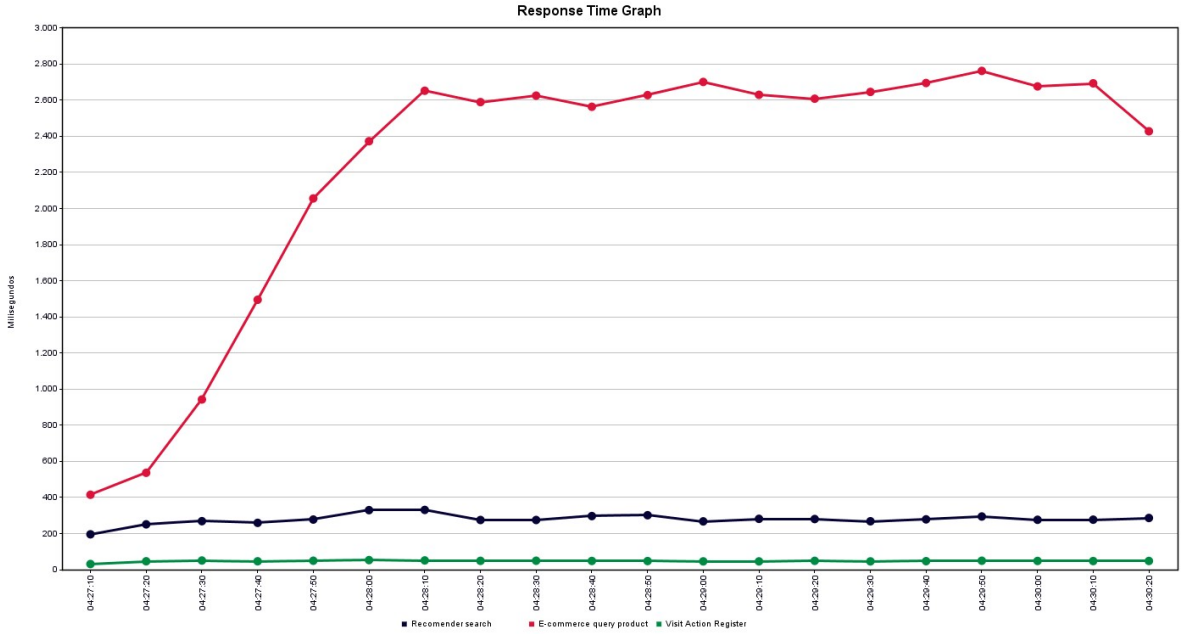


Figura 37: Gráfico de tiempo en segundos ejecutados versus hilos
Fuente: Resultados de JMeter.

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Estándar
Recomender search	3274	281	164	632	56,08
E-commerce query ...	3268	2293	215	10096	1377,73
Visit Action Register	3224	47	25	128	11,81
Total	9766	877	25	10096	1286,02

Etiqueta	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de Bytes
Recomender search	0,00%	16,4/sec	6,07	25,39	380,0
E-commerce query ...	0,00%	16,2/sec	361,38	25,08	22784,9
Visit Action Register	0,00%	16,3/sec	7,80	27,09	490,0
Total	0,00%	48,5/sec	374,46	76,84	7913,7

Figura 38: Métricas de resultados de caso de 6 instancias para servicio *e-commerce*
Fuente: Resultados de JMeter.

Los resultados de la prueba para el caso donde se esta corriendo el servicio *e-commerce* con 6 instancias y uso de cache:

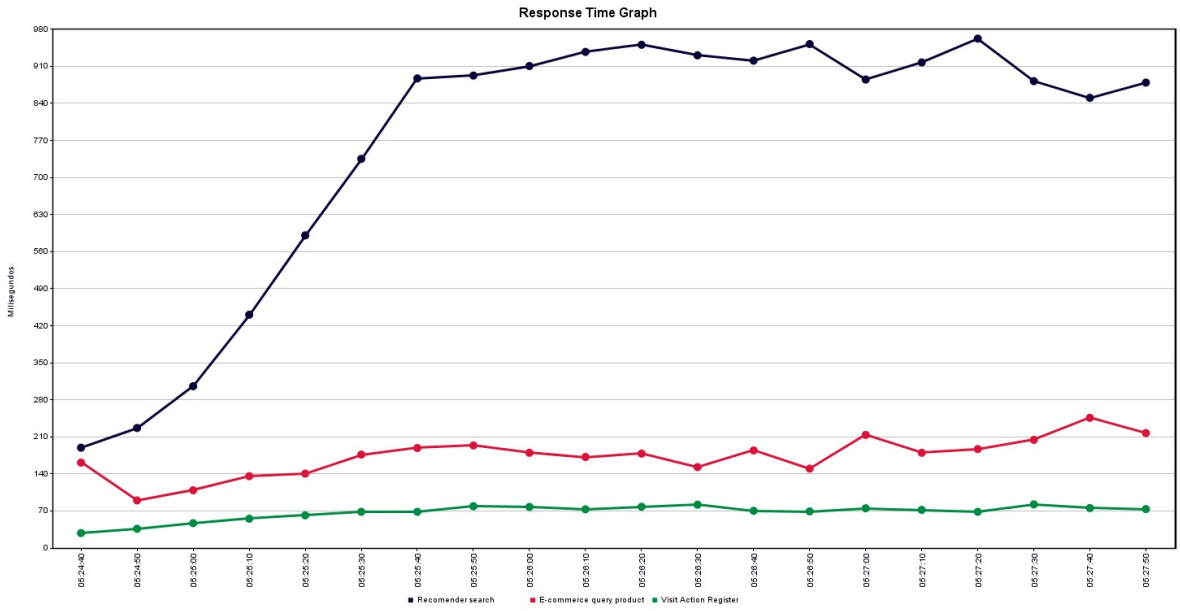


Figura 39: Gráfico de número de consultas para caso de 6 instancias con uso de cache para servicio *e-commerce*

Fuente: Resultados de JMeter.

Etiqueta	# Muestras	Media	Mín	Máx	Desv. Estándar
E-commerce query p...	8063	177	54	2102	117,43
Recomender search	8092	808	162	2015	276,87
Visit Action Register	8046	69	23	371	31,68
Total	24201	352	23	2102	370,04

Etiqueta	% Error	Rendimiento	Kb/sec	Sent KB/sec	Media de Bytes
E-commerce query p...	0,00%	40,3/sec	896,34	62,21	22779,1
Recomender search	0,00%	40,4/sec	15,00	62,72	380,2
Visit Action Register	0,00%	40,5/sec	19,37	67,27	490,1
Total	0,00%	120,8/sec	929,30	191,57	7879,3

Figura 40: Métricas de resultados de caso de 6 instancias con cache habilitado para servicio *e-commerce*

Fuente: Resultados de JMeter.

4.2.2. Análisis de resultados

Las métricas principales a considerar son el tiempo de demora en las consultas en los gráficos en función de la hora de ejecución, por otro lado, en las tablas son los valores de rendimiento, el porcentaje de error y el promedio de los tiempos de respuesta de cada servicio o total.

Los resultados en la prueba con 1 instancia del servicio *e-commerce* 36 muestran una clara falta de recursos a asignar al proceso asociado a la única instancia que está ejecutándose del servicio *e-commerce*, mostrando una tasa de error de consultas total de 66.98 %, lo cual

es bastante alto para efectivamente satisfacer las demandas de la prueba de estrés. Algo interesante a considerar es que esto pasa a llevar a los otros servicios también, esto debe ocurrir debido a que como hay un solo proceso sobrecargado de consultas, el servidor trata de mantener este proceso vivo a toda costa pasando a llevar el uso de recursos para los demás procesos. Notar además que en el gráfico claramente se nota lo sobrecargada que está la instancia, las bajas en los tiempos de consulta mostrados en el gráfico 35 son debido a los errores de consulta que avisan que el servicio no está disponible desde el servidor web, los cuales son respuestas livianas comparado a la cantidad de datos de una consulta bien ejecutada en este caso, razón por la cual el rendimiento muestra ser más alto que los demás casos siendo que las consultas no se realizaron mayoritariamente con éxito.

Los resultados en la prueba con 6 instancias del servicio *e-commerce* 36 muestra mejores resultados, de hecho con 0 % de errores, sin embargo, con la máxima carga de consultas de la prueba de estrés, las consultas llegan a durar alrededor de 2.6 segundos para el servicio de *e-commerce* según el gráfico 37, lo cual es bastante para una consulta para obtener solo un producto.

Los resultados en la prueba con 6 instancias del servicio *e-commerce* 40 muestra aún mejores resultados, habiendo consultas que duran una media máxima de 808 milisegundos en el caso de la consulta de búsqueda del sistema recomendador. El uso de cache permite aliviar la carga de la base de datos guardando los resultados de ciertas consultas en el cache del servicio de *e-commerce*. Notar además que el rendimiento es mucho mayor que en el caso anterior siendo de 120.8 consultas por segundo en total comparado a las 48.5 consultas por segundo de la configuración anterior.

El uso de cache y sus resultados comparados al segundo caso, da a entender que puede llegar a ser necesario revisar los recursos utilizados por la base de datos, o migrarlo ya sea o a una máquina con recursos dedicados a este o a alguna solución *cloud* para soportar mayores cargas de consultas.

Considerando el parámetro inicial de que en el lanzamiento la aplicación debe soportar 50 usuarios concurrentes dentro de la aplicación entonces, se podría considerar que eso está cumplido con la prueba de carga recién realizada.

CAPÍTULO 5
ESTADO DEL ARTE Y RETROSPECTIVA

Existen varios trabajos de diseño e implementación de arquitecturas de software e infraestructura basados en microservicios. Es más, actualmente muchas compañías tienden a migrar a este patrón para desplegar sus servicios como por ejemplo *Netflix* [Yury Izrailevsky, 2016] una aplicación de transmisión de series y peliculas, y *Uber* [Haddad, 2015] [Reinhold, 2016] una aplicación de locomoción particular de pasajeros.

El año 2020 se realizó una encuesta a 669 expertos de alrededor del mundo, respecto a prácticas y uso de tecnologías para desarrollar y desplegar esta arquitectura de software, cuyo resultado está en el libro “State of Microservices 2020” [Patryk Mamczur, 2020]. En lo que sigue, se mostrarán los resultados de la encuesta pertinentes al presente trabajo caracterizando aquellas practicas más populares en la industria. Además, analizamos cómo estas se utilizaron en el presente trabajo a modo de retrospectiva. Posteriormente, se revisará un par de ejemplos de estos despliegues y finalmente se hará una comparación de estos con la implementación llevada a cabo en el presente trabajo.

5.1. Resultados de encuesta versus decisión de uso en proyecto

5.1.1. Lenguaje de programación principal utilizado

Para el tiempo en que se realizó la encuesta, se mostró una tendencia a tecnologías que utilicen el lenguaje **Javascript** junto a **Typescript**, incluso se menciona que el 26 % de los expertos entrevistados utilizaron netamente esta tecnología.

En el presente proyecto se dio prioridad al lenguaje Python principalmente por la comodidad para agregar los algoritmos del sistema recomendador como se mencionó en la sección de propuesta de solución.

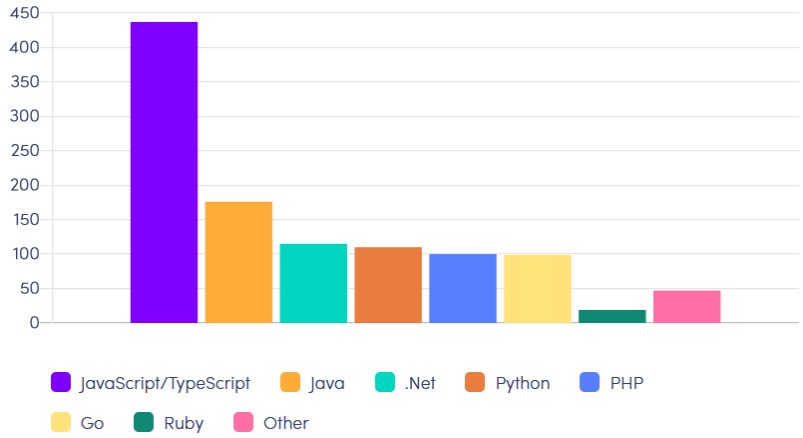


Figura 41: Preferencia de uso de lenguajes de programación para desarrollar microservicios
Fuente: `https://tsh.io/state-of-microservices/#programming-languages`

5.1.2. Despliegue y *Serverless*

En las formas de despliegue en la industria la mayoría prefiere por lo general tecnologías *cloud*, siendo el 34 % de los expertos los que mencionaron usar tecnologías *on premise*, lo cual se comprende debido a la robustez que ofrecen las tecnologías en la nube y lo caro que es tener un conjunto de servidores propio comparado a pedir instancias o máquinas virtuales a través de la nube a largo plazo.

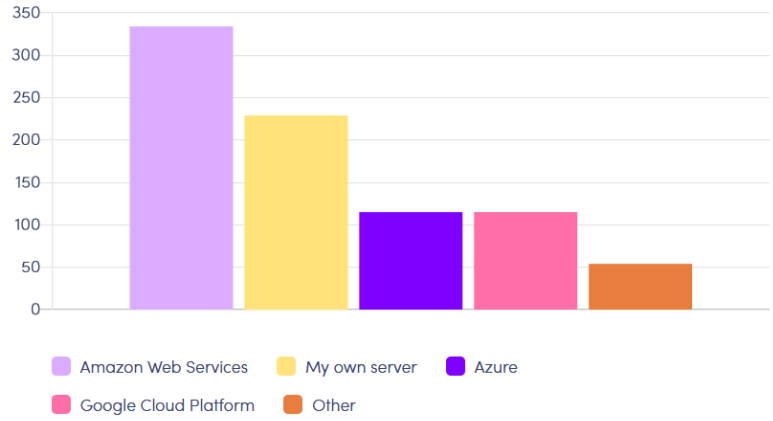


Figura 42: Preferencia de uso de herramientas de despliegue
Fuente: `https://tsh.io/state-of-microservices/#deployment-and-serverless`

Por otro lado, las tecnologías *Serverless* van ganando terreno en el despliegue de microservicios pero aún es una tecnología nueva que debe ser explorada por muchas empresas.

En el presente proyecto se utilizó mayoritariamente los servidores del departamento de informática de la universidad, debido a que como se mencionó antes, el departamento de informática otorga infraestructura para proyectos asociados a este.

Se pudo por ejemplo también usar tecnologías *cloud* para pedir máquinas virtuales como las que usamos del departamento de informática, sin embargo se obtendría lo mismo pero con la desventaja de que se tendría que pagar por las máquinas virtuales pedidas.

El gasto monetario empeora mucho más si se utiliza los servicios *cloud* de orquestadores que se ofrecen en el mercado, pues su uso es muy caro para los fondos que se tienen y con esos fondos se podría solamente usar por a lo más un par de meses las tecnologías de orquestación.

Es por esto que se decidió utilizar tecnologías *cloud* para *backing services* clave que requieran la robustez que ofrecen estos servicios, como en el caso particular de este proyecto, para la persistencia de imágenes del servicio de *e-commerce*.

5.1.3. Repositorios

Curiosamente existe un porcentaje no menor de expertos (32.9 %) que guarda los proyectos de microservicios en un solo gran repositorio, en donde se tienen los proyectos divididos en carpetas siendo que el estándar usual es tener un repositorio por servicio, una de las compañías que realiza esto es Google que guarda todos los proyectos en un solo gran repositorio, otras empresas que hacen lo mismo al menos hasta cierto punto son Microsoft, Facebook, Digital Ocean, Twitter y Uber. No obstante, esta forma de controlar versiones de los proyectos de microservicios esta en modo experimental en pequeñas compañías, algunas de las ventajas de esto es que facilita el uso de tecnologías de integración continua e integración entre los servicios que están dentro del repositorio.

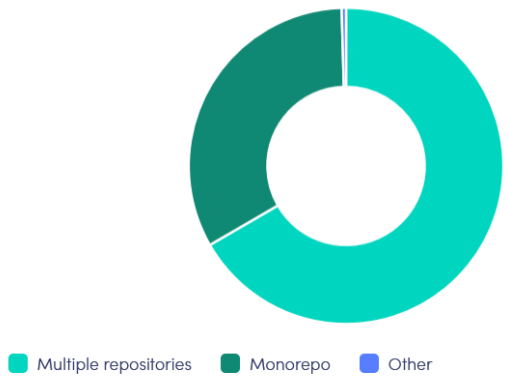


Figura 43: Preferencia de uso de repositorio/s para microservicios
Fuente: `https://tsh.io/state-of-microservices/#repositories`

Dentro de este proyecto se utilizó un repositorio por cada servicio desarrollado, esto para dar autonomía a los desarrolladores para desarrollar cada servicio y para evitar conflictos de edición por el control de versiones en el peor de los casos y ordenar mejor la documentación.

5.1.4. Comunicación, Autorización, intermediarios de mensaje

Sobre los protocolos de comunicación usados por la mayoría, prima el uso de HTTP para la comunicación entre microservicios.

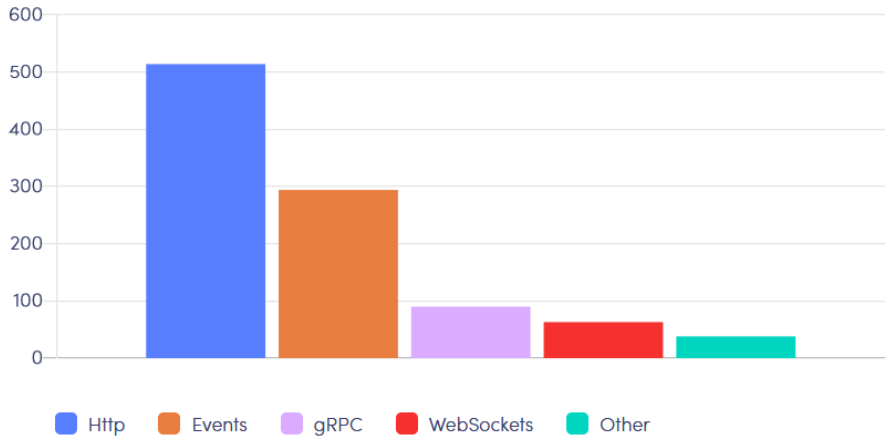


Figura 44: Preferencia de uso de protocolos de comunicación entre microservicios
Fuente: `https://tsh.io/state-of-microservices/#varia`

En este proyecto se usó HTTP pero para los casos en que se necesite realizar transacciones atómicas que impliquen el uso de otro/s servicios se utilizó RPC (*Remote Procedure Call*) como se mencionó anteriormente para asegurar que la transacción se mantenga consistente en esos casos.

Sobre cuando autorizar el usuario a realizar consultas por datos a un servicio, prima utilizar la misma *API Gateway* para autorizar las consultas pertinentes usando el proveedor de identidad.

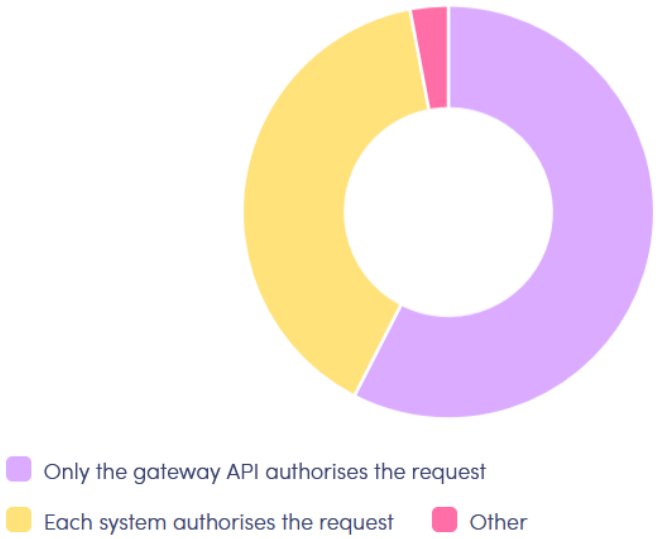


Figura 45: ¿Como se hacen cargo de autorizar las consultas de los servicios?
Fuente: `https://tsh.io/state-of-microservices/#varia`

En el presente proyecto no se utilizó el *API Gateway* para autorizar las consultas, en cambio se usó una plantilla de microservicio con un *middleware* implementado que consulte al proveedor de identidad si el *token* mandado es válido, negando la consulta si no lo es.

Esto se realizó debido a que hay un servicio en particular que no debe de tener autorización a través de uso de *token* para el uso de sus consultas, que es el servicio de manejo de *API* de administrador de proveedor de identidad, pues a través de este entre otras labores, se otorga un usuario invitado con rol y atribuciones limitadas cuyas credenciales son guardadas en la aplicación móvil. Sin embargo a este servicio se le configuró una *API Key*, es decir se pide una cadena especifica en la cabecera de la consulta para consultar al servicio, esto para evitar que cualquier cliente web pueda utilizar los recursos del servicio.

5.1.5. Depuración

Dentro de las herramientas de depuración de código, se considera el uso de *logs* como el más utilizado por los exponentes de la encuesta.

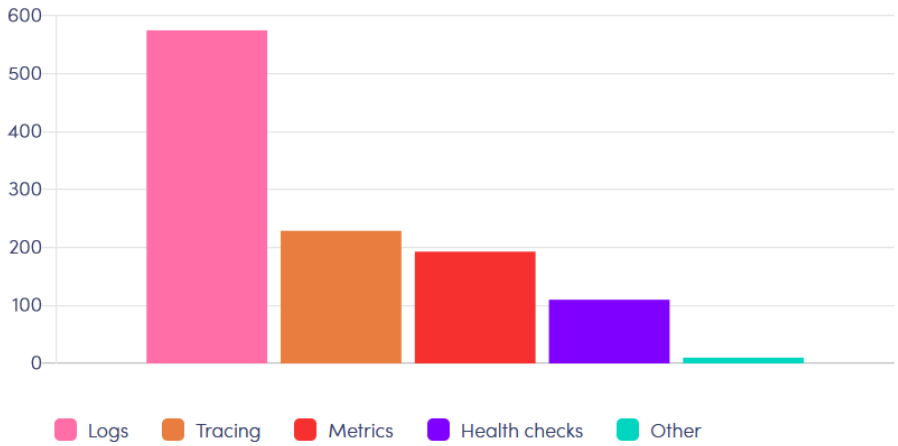


Figura 46: ¿Como se hacen cargo de autorizar las consultas de los servicios?
Fuente: `https://tsh.io/state-of-microservices/#varia`

En el presente proyecto no se utilizaron herramientas de *Quality Assurance* por qué al menos al día de hoy no surgió la necesidad, pero si el proyecto llega a crecer y se llega a implementar más microservicios, esto va a llegar a ser una necesidad, se hablará en más detalle de esto en las conclusiones.

5.2. Ejemplos de infraestructura y sus comparaciones

5.2.1. Uber

Uber, empezó utilizando una arquitectura monolítica la cual funcionaba bien al ser inicialmente una aplicación con la demanda de una sola ciudad, sin embargo cuando los servicios de Uber se hicieron disponibles a otras ciudades o a otros países, esto cambió, presentando problemas, entre estos destacan, que al crecer mucho la aplicación, las funcionalidades se hicieron demasiado acopladas al tener todo en un solo código base, dificultando la implementación de nuevas características, la integración continua se dificultó al tener que analizar todo el código de toda la aplicación por cada nuevo despliegue, estos entre otras cosas ocasionaron que deba de haber demasiada actividad de desarrollo entre resolver problemas, de la aplicación, solución de deudas técnicas y el añadido de nuevas características lo cual resulto a largo plazo demasiado difícil de mantener.

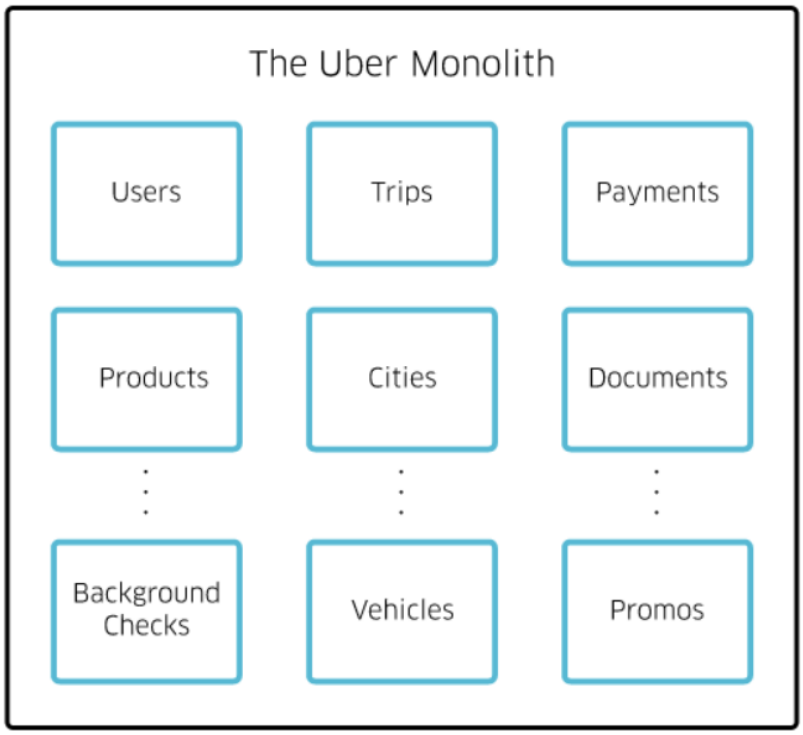


Figura 47: El monolito de Uber
Fuente: `https://eng.uber.com/service-oriented-architecture/`

Por tanto, decidieron desacoplar lentamente la aplicación monolítica en distintos servicios en lo que hoy se conoce como una arquitectura de microservicios, sin embargo esta no es labor fácil, el cambio de arquitectura de software implica mucho desarrollo y cambios en la infraestructura que requieren años de trabajo, siendo el objetivo final desacoplar convenientemente los servicios según el dominio de la aplicación.

La infraestructura mostrada en forma simplificada que se muestra en la siguiente figura 48 es en la que esta basada la arquitectura de microservicios que funciona el día de hoy en Uber.

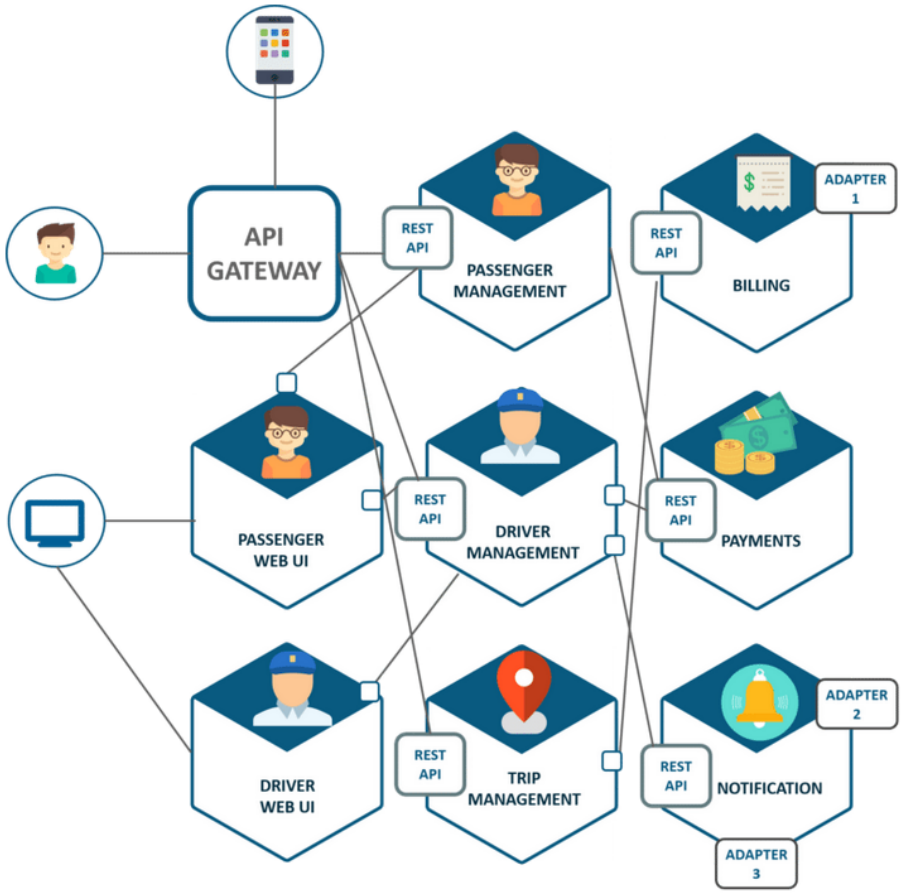


Figura 48: Ilustración de arquitectura de microservicios de Uber
Fuente: Sahiti Kappagantula
`https://www.edureka.co/blog/microservice-architecture/`

Uber además opera principalmente usando sus propios *datacenters* comparado a otras iniciativas que usan netamente tecnologías *cloud* para la infraestructura de los servicios desplegados.

Se observan principalmente que pasajeros y conductores se conectan a la *API Gateway* a través de la aplicación móvil para obtener los datos de la aplicación en su interfaz gráfica, eso aparte de la opción de interfaz web de pasajero y conductor que consume por su cuenta los servicios respectivos y accediendo a los servicios pertinentes a los casos de uso de la aplicación y estos son independientemente desplegables y escalables.

Uber usa orquestadores como Apache Mesos y Kubernetes y sus propia iteración sobre docker que es ucontainer , además construyeron su propio constructor de imágenes de contenedores llamado Makisu [Evelyn Liu, 2018]

5.2.2. Netflix

El caso de Netflix es muy parecido al de Uber, la aplicación, comenzó siendo diseñada como una arquitectura de software monolítica, sin embargo, fue una corrupción en la base de datos en Agosto del 2008 que llevó a Netflix grandes perdidas lo que les hizo tomar la decisión de migrar la arquitectura de la aplicación a una de microservicios desacoplados entre si, proceso que les tomo 7 años en terminar de migrar.

Aparte, decidieron migrar los despliegues de los servicios a despliegues en *cloud*, específicamente a *Amazon Web Services*, esto debido a la amplia gama de servicios y soluciones de despliegue que este ofrece, además de la dificultad de adaptar los recursos de una infraestructura *on premise* para cumplir con transmitir videos a millones de personas, es por esto que requirieron una infraestructura elástica, es decir, que adapte los recursos de procesamiento y almacenamiento a la demanda de usuarios.

Netflix depende de servicios *cloud* para todos los servicios escalables imaginables como lógica de negocios, bases de datos distribuidas, análisis de *big data*, recomendaciones, etc.

El día de hoy Netflix tiene cientos de microservicios desplegados en contenedores a través de su propio manejador de contenedores llamado Tituz [Amit Joshi, 2018] el cual al día de hoy es *open source*

5.2.3. Comparación con casos de implementación estudiados

Lo más interesante de lo observado en estos casos recién mencionados es la cantidad de servicios que estos tienen desplegados comparado a los del presente proyecto que son 9, si bien la estructura de las aplicaciones y los patrones por lo general coinciden, en el caso Netflix la diferencia está en el uso neto de tecnologías *cloud* lo cual es una decisión que tiene bastante sentido considerando la capacidad de recursos que requiere para servir las transmisiones a millones de personas. Por otro lado Uber usa herramientas de orquestación de contenedores como Apache Mesos y Kubernetes, lo cual comparado con el despliegue que se tiene en este proyecto son soluciones más solidas y de alta disponibilidad pero complejas a la vez.

Por otro lado, ambas aplicaciones han tenido que realizar sus propias soluciones de manejo de contenedores para que calcen de mejor manera con el flujo de trabajo que estos cumplen.

CAPÍTULO 6
CONCLUSIONES

Volviendo a la definición de requisitos no funcionales que se mencionó al principio de este trabajo de memoria, se analizará brevemente qué decisiones e implementaciones ayudan a cumplir estos, en el tema de la seguridad, se implementaron los protocolos necesarios, en particular el uso de *access token* junto al protocolo **OpenID Connect** además de la implementación del protocolo **TLS**. En rendimiento se implementan los protocolos HTTP 2.0 y uso de cache en los servicios desplegados para y escalabilidad, se da oportunidad para manipular apropiadamente según demanda de consultas la cantidad de instancias de los servicios y de poder añadir funcionalidades o servicios nuevos de ser necesario a futuro. Y en mantenibilidad, se cumplió en la mejor medida posible con la metodología de la aplicación de 12 factores.

Sin embargo en el desarrollo de aplicaciones no existe solución perfecta alguna para el producto final de una aplicación, solo ventajas y desventajas que hay que analizar en función de las limitantes que se tienen para la implementación de los distintos tipos de arquitectura de software e infraestructura, aplicando esta linea de pensamiento en el presente trabajo, una desventaja que se encontró dentro de la arquitectura de software basado en microservicios implementado es que los 6 desarrolladores *back-end* dentro del proyecto se tenía en un principio muy arraigada la arquitectura monolítica en la forma de pensar y solucionar problemas, por lo tanto hubo algo de perdida de tiempo en investigar y aplicar buenas prácticas.

Al día de hoy, la lógica de negocios de la aplicación Aliadas, está compuesta por 7 servicios asociados al dominio de este, siendo estos, el servicio de proveedor de identidad, chat, *e-commerce*, registro de acciones, búsqueda, captura de productos y el sistema de recomendación.

Si bien las pruebas indican que la arquitectura de infraestructura es capaz de aguantar el uso que le dará a la aplicación a los usuarios que el día de hoy están registrados, eso no asegura que a futuro esto siga funcionando, en particular por los finitos recursos utilizados para el despliegue de la aplicación, por tanto hay que estar atentos al uso de la aplicación por parte de los usuarios para ir a futuro buscando alternativas para cuando la infraestructura no soporte apropiadamente la demanda de usuarios sobre la aplicación.

Aparte, si la aplicación llega a crecer y se realizan nuevos cambios y por lo tanto nuevos servicios, hay que ir revisando lo que han hecho las industrias para manejar estos servicios, como por ejemplo y en particular los revisados en la sección del estado del arte que no manejan una cantidad de servicios que se pueda contar con los dedos de las manos, sino que cientos de servicios que se deben de mantener, desplegar y escalar apropiadamente y que en caso de que se use solamente **docker compose** para ello, que es lo que se esta usando en el presente proyecto, queda pequeño para el caso de uso de mantener esa gran cantidad de servicios. Algo rescatable es que con la forma en que se empezó a desarrollar el *back-*

end de la aplicación, se ignora el paso extremadamente complejo de migrar la arquitectura de software de un monolito a uno basado en microservicios como tuvieron que hacer los exponentes estudiados en el estado del arte.

La solución propuesta e implementada en este trabajo de memoria en el caso de la arquitectura de software, da el pie inicial para permitir que estos servicios y los que estén por venir a ser desarrollados si el proyecto crece aún más, sean fáciles de escalar apropiadamente, lo cual es en gran parte el objetivo de este trabajo.

Aún así, queda trabajo futuro si es que el uso de la aplicación llegase a crecer a niveles que la infraestructura actual no pueda soportar. Ideas que surgen después de lo aprendido con este trabajo son varias, entre ellas y si es que se quiere seguir utilizando la infraestructura del departamento de informática de la universidad, el uso de un orquestador de contenedores como **Kubernetes** con equipos dedicados a correr los servicios para ofrecer alta disponibilidad y la elasticidad que la infraestructura actual no ofrece, pues un orquestador se encarga también de escalar los servicios automáticamente según demanda de los usuarios.

En caso de que se tengan ingresos para el proyecto, entonces se podría usar la segunda idea que hay al respecto que es usar tecnologías *cloud* como **Amazon EKS** para levantar y orquestar los contenedores de las aplicaciones desarrolladas para este proyecto. O lo otro es que en lugar de ejecutar los servicios desarrollados en tecnologías *cloud*, se suba solo la persistencia de los datos del proyecto, lo cual puede ser más sabio pues ahí se dejaría de preocupar por mantener los *backing services on premise* que se tienen al día de hoy además de que en esos mismos servicios *cloud* ofrecen herramientas de respaldo automático usualmente, ahí habría que realizar una cotización de que conviene más además de revisar que se gana con cada caso o en el mejor de los casos, subir todo a *cloud* es una opción también.

Otro factor a considerar que se podría dejar como trabajo futuro es la implementación de herramientas de integración continua, razón por la cual, este proyecto no cumple con un proceso *DevOps*, es decir de desarrollo y despliegue integrados debido a que el proceso de despliegue y diseño son totalmente separados. Las razones por la que no se implementó herramientas de integración continua fueron principalmente de tiempo y de tener que capacitar a los desarrolladores para el uso de estas tecnologías, incluyendo al autor de este trabajo.

ANEXOS

Documentación de tecnologías utilizadas

En esta sección estará la documentación de las tecnologías utilizadas en el presente proyecto de memoria.

- Requisitos funcionales de proyecto Aliadas: https://docs.google.com/spreadsheets/d/1g0eWj3gbDaQFo2ZgTU7wEq9oZnWu-1HkrrsD9akOVDE/edit?usp=sharing
- Sobre CentOS: https://www.centos.org/
- Documentación de Docker: https://www.docker.com/
- Documentación de Nginx: https://www.nginx.com/
- Documentación de Kong: https://konghq.com/kong/
- Documentación de Keycloak: https://www.keycloak.org/
- Documentación de Identity Brokering: https://www.keycloak.org/docs/12.0/server_admin/#_identity_broker
- Repositorio de adaptador de proveedor de identidad de Apple para Keycloak: https://github.com/BenjaminFavre/keycloak-apple-social-identity-provider
- Repositorio de adaptador de proveedor de identidad de Mercado Libre para Keycloak: https://github.com/Jonskadev/keycloak-mercadolibre
- Información sobre JSON Web Token: https://jwt.io/
- Documentación de RHEL: https://www.redhat.com/es/technologies/linux-platforms/enterprise-linux
- Documentación de Spree: https://guides.spreecommerce.org
- Página oficial de fundación OpenID: https://openid.net/
- Lista de tecnologías de OpenID: https://openid.net/developers/certified/
- Documentación de Rocket Chat: https://docs.rocket.chat/
- Documentación de FastAPI: https://fastapi.tiangolo.com/

- Documentación de workers en FastAPI: `https://fastapi.tiangolo.com/deployment/server-workers/`
- Sobre Open API `https://www.openapis.org/`
- Documentación de Ruby on Rails: `https://guides.rubyonrails.org/`
- Documentación de workers en Ruby on Rails: `https://github.com/puma/puma`
- Documentación de RabbitMQ: `https://www.rabbitmq.com/documentation.html`
- Documentación de Certbot: `https://certbot.eff.org/`
- Sobre Let's Encrypt: `https://letsencrypt.org/`
- Sobre Kubernetes: `https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/`
- Sobre Amazon S3: `https://aws.amazon.com/es/s3/`
- Sobre Amazon EC2: `https://aws.amazon.com/es/ec2/`
- Sobre Amazon EKS: `https://aws.amazon.com/es/eks/`
- Sobre Google Kubernetes Engine: `https://cloud.google.com/kubernetes-engine`
- Documentación de MongoDB: `https://docs.mongodb.com/`
- Documentación de Elasticsearch: `https://www.elastic.co/guide/index.html`
- Documentación de Postgres: `https://www.postgresql.org/docs/`
- Documentación de JMeter: `https://jmeter.apache.org/usermanual/index.html`

Conceptos generales

En la presente sección se darán a conocer conceptos generales que aparecerán dentro del trabajo de título, estos son cruciales en el diseño e implementación de la arquitectura de software e infraestructura propuesta.

Estos estarán en orden *alfabético*.

- Aplicación: Es un programa que esta destinado a realizar algúna labor en específico sin relacionar a esa labor a uno asociado al computador en si sino que a solucionar un problema del usuario.

- *API*: Del acrónimo (*Application Programming Interface*), es una interfaz de comunicación para modificar datos a través de otras aplicaciones dentro de un sistema.
- *API Gateway*: Es un servicio que es una herramienta de gestión de *API*s que se encuentra entre uno o varios clientes y varios servicios *back end*, permite abstraer la ubicación de los servicios permitiendo exponer solo uno para consultar distintos servicios a través de *API*s.
- Contenedor: Los contenedores son unidades estándar que empaquetan código para poder desplegarlo y portarlo a cualquier sistema operativo.
- Cliente: Es un software o hardware (en este trabajo, software principalmente) que consume o accede a un servicio disponible a través de un servidor.
- Elasticidad: Capacidad de la infraestructura en adaptar los recursos requeridos en función de la demanda.
- Escalabilidad: La escalabilidad de una aplicación se define como la capacidad de éste de poder incrementar su capacidad y funcionalidades según demanda de usuarios sin comprometer su desempeño.
- *Framework*: Un *framework* es una estructura de soporte tecnológico y conceptual definido ya con módulos para solucionar un problema recurrente. Bajo el contexto de el presente trabajo, se referirá a *framework* a un *framework* web, osea, para solucionar problemas de aplicaciones web.
- Máquina Virtual: Es un software (usualmente un sistema operativo) que simula un sistema de computación y puede ejecutar programas como si fuese un computador real y físico.
- *Micro-Framework*: Es una informalidad para referirse principalmente a *frameworks* web que solo tienen la lógica de negocios, sin las vistas por ejemplo.
- Microservicio: Es un servicio que es parte de una funcionalidad de una aplicación desplegada. Esta es aislada del resto de los servicios que componen la aplicación y es desplegada independientemente, la diferencia con el concepto de servicio es que un microservicio abarca una parte acotada del dominio del sistema completo.
- Middleware: En el contexto del presente trabajo un *middleware* es una función que se ejecuta cada vez que llega una consulta antes de procesar esta.
- Servicio: Es una aplicación que realiza labores de manera automática, responde a eventos o escucha solicitudes de datos.
- Servicio *Back end*: Es la parte de una aplicación dedicada al manejo de datos y persistencia.
- Servicio *Front end*: Es la parte de una aplicación dedicada a la capa de presentación, es la parte con la que el usuario interactúa directamente.

- *Worker*: Una vez inicializados permiten ejecutar múltiples instancias de una aplicación para permitir paralelizar procesos idealmente en distintos núcleos del procesador donde está desplegado el servicio.

REFERENCIAS BIBLIOGRÁFICAS

[Abbott M. L., 2010] Abbott M. L., F. M. T. (2010). The art of scalability. scalable web architecture, processes and organizations for the modern enterprise, introduction to the akf scale cube. *Addison Wesley*, 1:325–338.

[Amit Joshi, 2018] Amit Joshi, e. a. (2018). Titus, the netflix container management platform, is now open source. https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436. Visitado en 15/1/2021.

[Evelyn Liu, 2018] Evelyn Liu, e. a. (2018). Introducing makisu: Uber's fast, reliable docker image builder for apache mesos and kubernetes. https://eng.uber.com/makisu/. Visitado en 15/1/2021.

[Fowler, 2019] Fowler, M. (2019). Software architecture guide. https://martinfowler.com/architecture. Visitado en 10-01-2021.

[Haddad, 2015] Haddad, E. (2015). Service-oriented architecture: Scaling the uber engineering codebase as we grow. https://eng.uber.com/service-oriented-architecture/. Visitado en 10/1/2021.

[IIBA, 2015] IIBA (2015). A guide to the business analysis body of knowledge, techniques. 1(1°):302–305.

[Kanjilal, 2021] Kanjilal, J. (2021). Securing modern e-commerce applications in 2021. https://resources.fabric.inc/blog/ecommerce-security. Visitado en 18/1/2021.

[M., 2015] M., R. (2015). Software architecture patterns. 1(3°):*.

[Patryk Mamczur, 2020] Patryk Mamczur, Tomasz Czechowski Mateusz Mól, M. N. (2020). State of microservices 2020. https://tsh.io/state-of-microservices/. Visitado en 10/1/2021.

[Reinhold, 2016] Reinhold, E. (2016). Rewriting uber engineering: The opportunities microservices provide. https://eng.uber.com/building-tincup-microservice-implementation/. Visitado en 10/1/2021.

[Richardson, 2019a] Richardson, C. (2019a). Microservices patterns with examples in java, deployment patterns. 1(1°):383–427.

[Richardson, 2019b] Richardson, C. (2019b). Microservices patterns with examples in java, escaping monolithic hell. *Manning*, 1(1°):14–20.

[Richardson, 2019c] Richardson, C. (2019c). Microservices patterns with examples in java, escaping monolithic hell. *Manning*, 1(1°):354–360.

[Richardson, 2021] Richardson, C. (2021). Decompose by business capability. https://microservices.io/patterns/decomposition/decompose-by-business-capability.html. Visitado en 18/1/2021.

[Wiggins, 2017] Wiggins, A. (2017). The 12 factor app. https://12factor.net. Visitado en 03-05-2021.

[Yury Izrailevsky, 2016] Yury Izrailevsky, Stevan Vlaovic, R. M. (2016). Completing the netflix cloud migration. https://about.netflix.com/en/news/completing-the-netflix-cloud-migration. Visitado en 4/1/2021.