Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Estudios de Postgrado
Maestría en Tecnología de la Información y la Comunicación

UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMALA

Ing. Marlon Alfredo Manzo Iboy
Asesorado por el Maestro Ing. José Andrés Lemus Arriaga

Guatemala, marzo de 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMALA

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ING. MARLON ALFREDO MANZO IBOY
ASESORADO POR EL MAESTRO ING. JOSÉ ANDRÉS LEMUS ARRIAGA

AL CONFERÍRSELE EL TÍTULO DE

MAESTRO EN TECNOLOGÍA DE LA INFORMACIÓN Y LA COMUNICACIÓN

GUATEMALA, MARZO DE 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Kevin Vladimir Cruz Lorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
DIRECTOR	Mtro. Ing. Edgar Darío Álvarez Cotí
EXAMINADOR	Mtro. Ing. Marlon Antonio Pérez Türk
EXAMINADOR	Mtro. Ing. Edwin Estuardo Zapeta Gómez
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMALA

Tema que me fuera asignado por la Dirección de la Escuela de Estudios de Postgrado, con fecha 11 de agosto de 2021.

Ing. Marlon Alfredo Manzo Iboy

Decanato
Facultad de Ingeniería
24189101- 24189102
secretariadecanato@ingenieria.usac.edu.gt

LNG.DECANATO.OI.151.2022

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Estudios de Posgrado, al Trabajo de Graduación titulado: **UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMA,** presentado por: **Marlon Alfredo Manzo Iboy ,** que pertenece al programa de Maestría en artes en Tecnologías de la información y la comunicación después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANA
FACULTAD DE INGENIERÍA

Inga. Aurelia Anabela Cordova Estrada

Decana

Guatemala, marzo de 2022

AACE/gaoc

Guatemala, marzo de 2022

LNG.EEP.OI.151.2022

En mi calidad de Director de la Escuela de Estudios de Postgrado de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor, verificar la aprobación del Coordinador de Maestría y la aprobación del Área de Lingüística al trabajo de graduación titulado:

"UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMA"

presentado por **Marlon Alfredo Manzo Iboy** correspondiente al programa de **Maestría en artes en Tecnologías de la información y la comunicación** ; apruebo y autorizo el mismo.

Atentamente,

"Id y Enseñad a Todos"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
DIRECCIÓN
ESCUELA DE POSTGRADO

Mtro. Ing. Edgar Darío Álvarez Cotí
Director
Escuela de Estudios de Postgrado
Facultad de Ingeniería

Guatemala, 14 de noviembre 2021

M.A. Edgar Darío Álvarez Cotí
Director
Escuela de Estudios de Postgrado
Presente

M.A. Ingeniero Álvarez Cotí:

Por este medio informo que he revisado y aprobado el **TRABAJO DE GRADUACIÓN** titulado: ”UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMALA, GUATEMALA” del estudiante **Marlon Alfredo Manzo Iboy** quien se identifica con número de carné **200313178** del programa de **Maestría en Tecnologías de la Información y la Comunicación.**

Con base en la evaluación realizada hago constar que he evaluado la calidad, validez, pertinencia y coherencia de los resultados obtenidos en el trabajo presentado y según lo establecido en el *Normativo de Tesis y Trabajos de Graduación aprobado por Junta Directiva de la Facultad de Ingeniería Punto Sexto inciso 6.10 del Acta 04-2014 de sesión celebrada el 04 de febrero de 2014.* Por lo cual el trabajo evaluado cuenta con mi aprobación.

Agradeciendo su atención y deseándole éxitos en sus actividades profesionales me suscribo.

Atentamente,

MARLON ANTONIO PEREZ TURK
INGENIERO EN CIENCIAS Y SISTEMAS
COLEGIADO No. 4492

MA. Ing. Marlon Antonio Pérez Türk
Coordinador
Maestría en Tecnologías de la Información y la Comunicación
Escuela de Estudios de Postgrado

Guatemala, 16 octubre de 2020.

M.A. Ing. Edgar Darío Álvarez Cotí

Director

Escuela de Estudios de Postgrado

Presente

Estimado M.A. Ing. Álvarez Cotí

Por este medio informo a usted, que he revisado y aprobado el Trabajo de Graduación y el Artículo Científico: "**UNA SOLUCIÓN DE SOFTWARE QUE GARANTIZA LA NO DEGRADACIÓN EN EL TIEMPO DE EJECUCIÓN DE LOS ELEMENTOS DEL SISTEMA EN UNA ARQUITECTURA DE MICROSERVICIOS CON AYUDA DE SOLUCIONES OPEN SOURCE PARA MONITOREO EN LA NUBE DE MICROSOFT AZURE, EN LA CIUDAD DE GUATEMALA, GUATEMALA**" del estudiante **Marlon Alfredo Manzo Iboy** del programa de Maestría en **Tecnología de la Información y de la Comunicación**, identificado con número de carné: **200313178.**

Agradeciendo su atención y deseándole éxitos en sus actividades profesionales me suscribo.

José Andres Lemus
Ingeniero en Sistemas
Colegiado 12305

Ing. M.Sc. José Andrés Lemus Arriaga

Colegiado No. 12305

Asesor de Tesis

ACTO QUE DEDICO A:

Dios	Por haberme dado la vida. Por sus inmensas bendiciones, que son nuevas cada día.
Mis padres	Humberto Manzo y Maria Luisa Iboy. El amor de ellos será para siempre mi inspiración.
Mi esposa	Karla Reyes de Manzo, por su apoyo incondicional en cada decisión que tomamos para seguir adelante.
Mi tío	Roberto Iboy (q.d.e.p.) ejemplo de vida y perseverancia, siempre te recordaremos en nuestros corazones.
Mi hermano	Christian Manzo, por ser una importante influencia para seguir adelante y nunca desmayar en el camino.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por darme la oportunidad de formarme como profesional.
Facultad de Ingeniería	Porque en sus aulas recibí el conocimiento necesario para aplicarlo exitosamente en la vida.
Mis amigos de la Maestría	Roberto García, Estuardo Noack, ejemplos de vida y compañerismo.
Dra. Aura Marina Rodríguez	Por compartir de una forma acertada sus conocimientos para la elaboración del trabajo de graduación
Mi asesor	Maestro. José Andrés Lemus Arriaga, por compartir de su conocimiento en la elaboración de este trabajo de graduación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
km	Kilómetros
m	Metros
m^2	Metros cuadrados
nm	Nanómetro

GLOSARIO

Docker Swarm	Es una herramienta integrada en el ecosistema de *Docker* que permite la gestión de un *cluster* de servidores
Microservicios	Son unidades funcionales concretas e independientes, que trabajan juntas para ofrecer la funcionalidad general de una aplicación.
Microsoft Azure	Es un servicio de computación en la nube creado por Microsoft para construir, probar, desplegar y administrar aplicaciones y servicios mediante el uso de sus centros de datos**.**
***Software* Monolítico**	Hace referencia a una aplicación *software* en la que la capa de interfaz de usuario y la capa de acceso a datos están combinadas en un mismo programa y sobre una misma plataforma.
Open Source	Es el *software* cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público.

Tecnología de la nube Conocida también como servicios en la nube, es un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es *internet*.

RESUMEN

El propósito de la investigación fue definir una solución a la problemática del monitoreo de sistemas basados en arquitecturas de microservicios montados en la nube de Microsoft Azure, de tal forma que al implementar una solución de *software* se encuentre constantemente registrando eventos anómalos y con esa información anticiparse a cualquier degradación en las operaciones.

El objetivo general del estudio consistió en el diseño e implementación de una solución de *software* que garantice la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

El estudio a esta problemática se basa en la recolección de información de forma constante sobre los estados en el comportamiento de cada contenedor que componen este tipo de arquitecturas. La metodología de investigación utilizada para esta investigación fue de tipo experimental debido a que se analizaron una base de datos de eventos del sistema para luego buscar un enfoque de tipo cuantitativo.

Se aplicaron las mejores prácticas dictadas por la documentación oficial de Microsoft Azure para resolver cada uno de los objetivos planteados en el trabajo de investigación. Para resolver el problema al monitoreo se configuraron soluciones *Open Sorce* para que operen en concordancia tales como, *InfluxDB*, *Telegraf*, *Kapacitor* y *Chronograf*.

La principal conclusión a este trabajo de investigación es que el monitoreo en sistema con arquitectura de microservicios es uno de los problemas más complejos que existen debido a que este tipo de soluciones entre más van creciendo, es más difícil seguir el comportamiento de sus elementos.

De acuerdo con los resultados obtenidos, se recomienda que estas soluciones y configuraciones expuestas a esta problemática, al implementar monitoreo automático se obtiene baja en la dependencia humana para la intervención de cualquier evento, incrementando la disponibilidad operativa del sistema y mejorando la capacidad mínima de recursos necesarios para operar en el sistema.

OBJETIVOS

General

Determinar una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

Específicos

- Establecer la configuración e interconexión de una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

- Determinar cómo se asegura la comunicación entre microservicios ante cualquier ataque de una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

- Establecer cómo se garantiza la atomicidad de un componente de microservicio ante eventos de caída en una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones

open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

RESUMEN DEL MARCO METODOLÓGICO

En esta sección se explican los detalles técnicos del documento de investigación, el cual lleva por nombre *Una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala,* el estudio de esta problemática se basa en la recolección de información de forma constante sobre los estados en el comportamiento de cada contenedor que componen este tipo de arquitectura.

Enfoque de la investigación

El tipo de estudio que se utilizó fue estadístico matemático, por sus características, es el que mejor aportó información a la investigación, el estudio de esta problemática se basó en la recolección de información de forma constante sobre los estados en el comportamiento de cada contenedor que componen este tipo de arquitectura, para luego realizar su respectivo análisis y con los resultados, se resolvió cada una de las preguntas principales y secundarias planteadas.

Tipo de la investigación

El tipo de investigación que se aplicó fue de tipo descriptivo, debido a que fue necesario analizar de forma individual cada contenedor, determinar causas y razones por las cuales uno de ellos puede dejar de funcionar totalmente.

Diseño de la investigación

Con la información recolectada y analizada, se realizaron pruebas y experimentos de forma empírica para luego volver a analizar el resultado y los posibles nuevos comportamientos de cualquier contenedor, con este ejercicio constante, se buscó predecir futuras bajas en el rendimiento de uno o varios microservicios y establecer un mejor uso y rendimiento de los recursos de cada uno de ellos, para esto se aplicó un tipo de diseño experimental.

Variables

Las variables estudiadas durante el proceso de esta investigación fueron: microservicios y contenedores, con el fin de obtener resultados para el análisis de la investigación. Las definiciones pueden observarse en la tabla I.

Operacionalización de variables

Con el fin de detallar como se operar y se debe de leer cada una de las variables que se trabajarán en el trabajo de investigación, se observa lo siguiente.

Tabla I. **Operacionalización de variables**

Problema	Variable	Definición	Dimensión	Indicador
Analizar e identificar el estado operacional que se encuentra el microservicio	Microservicio	Unidades funcionales concretas e independientes que trabajan juntas para ofrecer la funcionalidad general de una aplicación	Monitorización Comunicación Tolerancia a fallos	Latencia en la comunicación entre microservicios. Cantidad de elementos que componen la arquitectura.
Identificar si se tienen los recursos necesarios para que el microservicio opere correctamente	Contenedor	Alternativa de virtualización por *software* en la cual se crea la percepción de un ambiente aislado y exclusivo para aplicaciones dedicada a una función	Configuración	Tamaño en disco duro. Cantidad de memoria RAM. Cantidad de mensajes para enviar y guardar.

Fuente: elaboración propia.

Unidad de análisis

La población en estudio fue un conjunto de microservicios que componen un sistema configurado en la nube de Microsoft Azure, cada microservicio fue analizado de forma individual durante los meses de julio y agosto de 2021, registrando la información en una base de datos llamada *InfluxDB*, para luego estudiar su comportamiento y con ese análisis determinar posibles soluciones para corregir cualquier problema que pueda estar presentando.

Fases del estudio

Para la elaboración de este trabajo de investigación se utilizaron 5 fases de estudio, los cuales se describen a continuación:

- Fase 1. Revisión literaria: se realizó una investigación literaria para identificar qué trabajos de investigación existen que puedan apoyar con la realización del presente trabajo.

- Fase 2. Recolección de la información: se recolectaron los datos registrados de cada microservicio en análisis, la información recolectada corresponde a los meses de julio y agosto de 2021, las dimensiones en análisis fueron, bytes escritos y leídos disco duro, cantidad de memoria RAM utilizada y cantidad de paquetes enviados y recibidos a través de vía http.

- Fase 3. Análisis de la información: se analizaron cada uno de los resultados obtenidos, utilizando la herramienta para análisis de información *Power BI,* debido a que la cantidad de información recolectada superaba más de 2,000 registros por día, esta herramienta permitió aplicar gráficos de tendencias, histogramas de frecuencia, entre otros, y con ello facilitar la interpretación de los datos.

- Fase 4. Interpretación de la información: se interpretaron los resultados de la fase anterior, se aplicaron ajustes sobre cada una de las configuraciones de los microservicios con el fin de alcanzar cada uno de los objetivos planteados, después de aplicar los ajustes se regresaba a la fase anterior, “Análisis de la información”, hasta alcanzar los resultados deseados.

- Fase 5. Redacción del informe final: por último, después de tener toda la información literaria de apoyo y el análisis e interpretación de los datos, se procede con la redacción del informe final incluyendo cada una de las secciones que se solicita el normativo para la elaboración del informe final.

INTRODUCCIÓN

El presente trabajo de investigación expone la implementación de las mejores prácticas utilizadas para la configuración de sistemas basados en arquitecturas de microservicios. La investigación es una sistematización para la problemática central, una implementación de *software* que ayude al monitoreo de estos sistemas garantizando la no degradación de sus elementos en tiempo de ejecución utilizando soluciones *Open Source.*

La inexistencia de una solución de *software* que ayude en la administración y monitoreo de cada uno de los microservicios que componen estas arquitecturas, es la razón principal por la cual todavía este tipo de arquitectura no es muy popular y por ende su implementación y la documentación es un poco limitada, pero existen varios casos de éxito que apoyan estas arquitecturas por lo que esto ayuda a la toma de decisión para una implementación de este tipo.

La importancia de una implementación de *software* que apoye al monitoreo de cada uno de sus elementos radica en que los sistemas evolucionan constantemente, esto conlleva la necesidad de estar innovando en cada una de las soluciones de *software* que se tengan implementadas, desde este punto, es necesario considerar que estos sistemas también crecen en su complejidad operativa, es necesario presentar soluciones que ayuden a mejorar su rendimiento, la eficiencia de los recursos que los componen y garantizar su atomicidad en las operaciones que realizan.

El aporte del presente trabajo de investigación consistió en presentar las mejores prácticas y configuraciones en arquitecturas de microservicios para la implementación de una solución de *software* que garantice la no degradación en el tiempo de ejecución en estos sistemas, utilizando la nube de Microsoft Azure implementando soluciones *Open Source.*

La metodología de la investigación utilizada para esta investigación fue de tipo descriptivo debido a que se analizaron una base de datos de eventos del sistema para luego buscar un enfoque de tipo cuantitativo, se dejó constancia de cada una de las instrucciones que se utilizaron para la implementación de la solución, utilizando las mejores prácticas definidas en la documentación oficial de Microsoft Azure y con ello se logró registrar en una base de datos la información de los resultados.

En el primer capítulo se describe el marco teórico, forma parte base para la realización de todo el trabajo de investigación. Cada uno de los conceptos que forman parte del trabajo de investigación se pueden encontrar en esta sección como, por ejemplo: microservicios, monitoreo, *Open Source*, Microsft Azure, *Software*.

En el segundo capítulo se presentan los datos obtenidos como resultado a la implementación de la solución propuesta, la implementación de las mejores prácticas en la configuración de microservicios definidas por Microsoft y la propuesta a la solución al monitoreo en este tipo arquitecturas.

En el tercer capítulo se realiza la discusión de resultados, en el análisis interno se utilizó una comparativa porcentual en la mejora del sistema una vez implementado cada una de las propuestas planteadas y en el análisis externo de analizaron algunos trabajos de investigación que presentan objetivos semejantes a la problemática principal de este trabajo de investigación, los cuales fueron de gran ayuda para la definición de la solución propuesta.

.

1. ANTECEDENTES

En el presente capítulo se describen aspectos de como una evolución natural del desarrollo de *software*, ha evolucionado a un nuevo estilo de *software*, la arquitectura de microservicios. Cilleruelo (2016), describe este estilo de arquitectura como una arquitectura en constante evolución y con un futuro ciertamente muy prometedor, en la que hoy en día muchas empresas y profesionales ya están en investigación de una nueva manera de concebir y desarrollar *software* de calidad.

1.1. Generalidades

Brown (2016) en su trabajo de investigación presentado con el título: *Mas allá de clichés: Una breve historia de los patrones de microservicios* expone lo siguiente:

> La arquitectura de microservicios comenzó a obtener atención después de que una serie de historias de éxito fueron publicada por compañías como Netflix, Gilt.com y Amazon, entre otras, sin embargo, todas estas compañías provienen de compañías web, que se encuentran siempre en constante cambio para mejorar los servicios que proveen. Al momento de que las corporaciones tradicionales adoptan las arquitecturas de *software* basadas en microservicios, uno de los temas con los que se enfrentan son particularmente “La Gestión de datos Descentralizados” y “La Gobernanza Descentralizada”, es aquí donde se presenta la necesidad de un sistema de monitoreo (p.48).

1.1.1. Análisis de resultados de investigaciones previas

Los autores Ghofrani y Lübke, (2018), en su trabajo de investigación, *Challenges of Microservices Architecture: A Survey on the State of the Practice* plantearon los resultados de su trabajo como:

> Las arquitecturas basadas en microservicios surgieron aproximadamente en el año 2013, a partir de este año, que se han conocido algunas historias de éxito, por lo cual, esta arquitectura de *software* que se ha vuelto muy popular y algunos estudios que han demostrado su eficiencia y eficacia en el uso de los recursos de *software* (p.53).

1.1.1.1. Análisis a nivel internacional

En el estudio de redes para la interconexión de contenedores *Dockers*, define Zhang (2018) como:

> Un análisis en la comunicación y administración durante el crecimiento en la cantidad de contenedores que pueden ser administrados por *Dockers*, *Kubernetes, Weave*, etc. Se presenta como solución al problema que existe en la comunicación entre contenedores diferentes esquemas de conexión entre ellos, al plantear que esto puede ser unas de las principales causas de la falta de atomicidad en la información entre microservicios (p.16).

En la investigación, *Orquestación Automática de Contenedores*, realizado por Casalicchio (2015) expone:

> Hoy en día, los marcos de orquestación de contenedores están en su infancia y no incluyen ninguna característica autonómica, *Cloudify,*

Kubernetes, permiten la orquestación de contenedores *Dockers*, sin embargo, como ejecutar y orquestar en un entorno distribuido sin aprovechar completamente todos los recursos. *Docker Swarm* requiere una estática configuración de los nodos del *cluster.*

Se formulan las siguientes preguntas como resultado a este documento de investigación: ¿Por qué?, ¿Cuándo?, ¿Dónde? Y ¿Cómo? construir, cargar, poner en marcha, migrar y apagar los contenedores bajo políticas establecidas en los comportamientos registrados (p.2).

En el documento de investigación, *Método de automatización del Despliegue continuo en la Nube para la Implementación de Microservicios* según Vera (2016):

La arquitectura de microservicios llegó a cumplir las expectativas esperadas, sin embargo, aún se presentan desafíos como la complejidad de tener que gestionar pequeños sistemas distribuidos, la latencia de la red y la falta de fiabilidad, la tolerancia a fallas, la coherencia e integración de datos, la gestión de transacciones distribuidas, las capas de comunicación, el balanceo de carga, la orquestación, el monitoreo y la seguridad (p.4).

Cada uno de los elementos que plantea el autor “hacen resaltar la necesidad que existe en reforzar cada una de las tecnologías que se utilizan para cada uno de ellos, pero como saber cuándo cada uno de ellos podría dejar de operar adecuadamente, únicamente al tener un sistema de monitoreo bastante completo” (Vera, 2016, p.6).

Cruz, (2016) en la investigación *Gestión Eficiente de Arquitecturas Basadas en Microservicios* expone lo siguiente:

Una consecuencia del uso de microservicios como componentes, es que las aplicaciones necesitan ser diseñadas de manera que pueden tolerar fallos en el servicio. Cualquier llamada a un servicio podría fallar debido a la indisponibilidad del proveedor del servicio. El cliente tiene que responder a este evento de la mejor manera posible.

Como los servicios pueden fallar en cualquier momento, es importante ser capaces de detectar fallos lo más rápido posible, si es posible automatizar la restauración del servicio. Las aplicaciones de microservicios ponen mucho énfasis en la monitorización en tiempo real de la aplicación, se comprueban los elementos de la arquitectura y las métricas importantes y concluye con la importancia que existe en la automatización para la restauración de los servicios, por eso la necesidad que existe actualmente de un sistema de monitoreo que reaccione ante cualquier siniestro (p.47).

En el trabajo de investigación, *Un marco de Monitoreo y Análisis Nativo de la Nube* según explican los autores Oliveira, et. al. (2017) plantean lo siguiente:

La visibilidad operacional es una capacidad administrativa importante y es uno de los factores críticos para decidir el éxito o el fracaso de un servicio de voz alta. Hoy en día, se vuelve cada vez más complejo en muchas dimensiones, lo que incluye poder rastrear estados del sistema persistentes y volátiles, así como proporcionar servicios de niveles más altos, como análisis de registros, descubrimiento de *software*, detección de anomalías conductuales, análisis de deriva, por nombrar algunos.

Además, los puntos de destino para monitorear son cada vez más variados en términos de su heterogeneidad, cardinalidad y ciclos de vida, mientras se alojan a través de diferentes acumulaciones de *software* (p.1).

Uno de los factores críticos para poder decidir el éxito o fracaso de un sistema, es el nivel de visibilidad que se tiene en las operaciones, saber hacia dónde se dirige un sistema y con ello tomar las mejores decisiones, con esto la necesidad de un sistema de monitoreo avanzado (Manzo, 2021).

Los sistemas basados en arquitecturas de microservicios, con el tiempo, éstos tienden a crecer en funciones y por consiguiente, es necesario agregar más contenedores, el monitoreo del sistema también se vuelve cada vez más complejo, el rendimiento en cada uno de los contenedores se hace más recurrente, es necesario diseñar un sistema que basándose en el monitoreo de cada uno de los contenedores y con la información que se pueda reunir en un lapso de tiempo, que el sistema sea capaz de recuperarse en temas de rendimiento, sin tener que reiniciar completamente el servicio (López, 2017).

1.1.1.2. Análisis a nivel nacional

En el país de Guatemala, existen investigaciones respecto al uso y configuración de arquitecturas basadas en microservicios, se pueden mencionar cursos impartidos por NobleProg, ubicados en El Centro Europlaza Guatemala, en el segundo nivel de una de las cuatro torres que conforman el Europlaza World Business Complex, en la ciudad de Guatemala zona 14, estos cursos incluyen los niveles básicos, intermedio y avanzados, se garantiza al estudiante que obtendrá los conocimientos necesarios para montar un sistema completo basado en esta arquitectura.

En la tesis que se titula, *Diseño e implementación de una Arquitectura escalabra basada en Microservicios para un Sistema de gestión de Aprendizaje con características de red social*, publicada por la Universidad de San Carlos de Guatemala se expone lo siguiente:

El uso de herramientas para el monitoreo de los recursos utilizados por los contenedores de *software*, para el caso de contenedores Docker, se aconseja el uso de la herramienta de *software* libre *Cadvisor* de Google, la cual permite monitoreo de espacio de disco, uso de CPU y memoria RAM.

Por último, la recomendación en el análisis de otras herramientas como *Docker Sware o Kubernetes* de Google que, en desarrollo de sistemas muy grandes, permiten realizar de forma automática muchas de tareas concurrentes, pero en sistemas más grandes, el manejo de la infraestructura se complica (Paz, 2017, p.61).

De esta tesis se extrae como conclusiones que con el uso excesivo de los contenedores Docker para lograr el despliegue de las aplicaciones, este tipo de arquitecturas con forme van creciendo, la misma va requiriendo una configuración más compleja, sin embargo, este tipo de patrones de diseño de software orientados a microservicios, la gran ventaja es que cada módulo puede operar de manera independiente mejorando su eficiencia y tiempos de respuesta.

Todas las soluciones que utilizan arquitecturas basadas en microservicios tienen que utilizar una nueve de internet que pueda ofrecer este tipo de servicios y configuraciones, una de las nubes que ofrecen este tipo de servicios se puede mencionar *Amazon Web Services* (AWS) y en el trabajo de graduación titulado, *Diseño e Implementación de una Arquitectura en AWS que Garantice una Alta Disponibilidad de un Sistema Web Orientado a Microservicios para la Tabulación y Presentación de los Resultados de las Elecciones de Guatemala*, realizado por Luis, (2021) de la Universidad de San Carlos de Guatemala, utilizó patrones de diseño para mejorar el nivel de atomicidad en la información del sistema exponiendo lo siguiente:

> La arquitectura de replicación “Master-Slave” ayudó a independizar los sistemas de tabulación y presentación de resultados al permitir la consulta de los resultados desde una base de datos secundaria y el ingreso de estos desde la Base de Datos primaria, mejorando así la atomicidad de los datos (p. 115).

Durante la prueba de carga, la arquitectura en AWS que soporta el microservicio para la consulta de los resultados registró un nivel de disponibilidad del 99.95 %. Al mismo tiempo, la prueba de la API del sistema de tabulaciones obtuvo un nivel de disponibilidad del 100 % al procesar correctamente todas las solicitudes para obtener registros consistentes.

Del trabajo de graduación expuesto anteriormente se concluye que existen diversidad de patrones de diseño que pueden ayudar a garantizar la atomicidad de la información al implementar arquitecturas basadas en microservicios utilizando recursos de la nube de AWS, en comparativa de la nube de Microsoft Azure, se investigó un patrón de diseño equivalente al expuesto por Luis Alvarado, se encontró y se implementó el patrón de diseño SAGA, el cual se describe más a detalle en la sección de presentación de resultados.

2. MARCO TEÓRICO

En este capítulo se presenta cada uno de los conceptos teóricos fundamentales que forman parte del conocimiento en materia sobre los elementos que componen las arquitecturas de *software* basadas en microservicios como también las herramientas más utilizadas para su configuración y control.

2.1. Solución de s*oftware*

Fumio, (2001) en el documento de investigación *Metodología para la Definición de Software de una Manera Determinista,* plantea el concepto de *software* como:

> La aparición de *software* para sistemas informáticos se atribuye principalmente al progreso de la tecnología de desarrollo de *hardware*, pero no al desarrollo de la ingeniería de *software*. Esto se desprende claramente de la evidencia de que, desde los primeros días del desarrollo de *software*, el *software* se ha desarrollado de una manera como si hubieran intentado construir un edificio de gran altura. La funcionalidad como sistema, que se adquiere como resultado del desarrollo de *software*, se publicita, mientras que el tema del desarrollo de *software* en sí mismo siempre se olvida (p. 3).

2.1.1. Tipos de s*oftware*

Se pueden clasificar en categorías según la función, el tipo o el campo de uso comunes. Hay tres clasificaciones amplias:

- El *software* de aplicación es la designación general de los programas de computadora para realizar tareas. El *software* de aplicación puede ser de uso general (procesamiento de texto, navegadores web, entre otros) o tener un propósito específico (contabilidad, programación de camiones, entre otros). El *software* de aplicación contrasta con el *software* del sistema.

- El *software* del sistema es un término genérico que se refiere a los programas informáticos que se utilizan para iniciar y ejecutar sistemas informáticos, incluidos diversos programas y redes de aplicaciones.

- Las herramientas de programación informática, como los compiladores y el enlazador, se utilizan para traducir y combinar el código fuente y las bibliotecas de programas informáticos en RAM ejecutables (programas que pertenecerán a uno de los tres mencionados)

2.1.2. Metodologías

Para administrar un proyecto de manera eficiente, el gerente o el equipo de desarrollo debe elegir la metodología de desarrollo de *software* que funcionará mejor para el proyecto en cuestión. Todas las metodologías tienen diferentes fortalezas y debilidades y existen por diferentes razones. A continuación, se ofrece una descripción general de las metodologías de desarrollo de *software* más utilizadas.

- Metodología ágil: los equipos utilizan la metodología de desarrollo ágil para minimizar el riesgo (como errores, sobrecostos y requisitos cambiantes) al agregar nuevas funciones. Hay muchas formas diferentes del método de desarrollo ágil, que incluyen *scrum, crystal*, programación extrema (XP) y desarrollo basado en características (FDD).

- Implementación DevOps: no es solo una metodología de desarrollo, sino también un conjunto de prácticas que respaldan una cultura organizacional. La implementación de DevOps se centra en el cambio organizacional que mejora la colaboración entre los departamentos responsables de los diferentes segmentos del ciclo de vida del desarrollo, como el desarrollo, la garantía de calidad y las operaciones.

- Desarrollo en cascada: muchos consideran que el método en cascada es el método de desarrollo de *software* más tradicional. El método en cascada es un modelo lineal rígido que consta de fases secuenciales (requisitos, diseño, implementación, verificación, mantenimiento) que se centran en objetivos distintos. Cada fase debe estar completa al 100 % antes de que pueda comenzar la siguiente. Por lo general, no existe un proceso para volver atrás para modificar el proyecto o la dirección.

- Desarrollo rápido: el desarrollo rápido de aplicaciones (RAD) es un proceso de desarrollo condensado que produce un sistema de alta calidad con bajos costos de inversión. El método de desarrollo rápido de aplicaciones consta de cuatro fases: planificación de requisitos, diseño de usuario, construcción y transición. Las fases de diseño y construcción del usuario se repiten hasta que el usuario confirma que el producto cumple con todos los requisitos.

2.2. Elementos de un sistema de *software*

Un sistema de *software* tiene tres tipos básicos: programas de aplicación, controladores de dispositivos y sistemas operativos. Cada tipo de *software* realiza un trabajo completamente diferente, pero los tres trabajan en estrecha colaboración para realizar un trabajo útil.

Si bien algunos programas de propósito especial no encajan perfectamente en ninguna de estas clases, la mayoría del *software* sí lo hace. Los programas se ejecutan en la parte de la memoria del sistema. Mientras se ejecutan, los programas se conocen como procesos o trabajos.

2.2.1. Tipos de elementos de *software*

A continuación, se describen los tres elementos básicos que forman parte de un sistema de *software*:

- Programas de aplicación: son la capa superior de *software*. Puede realizar tareas específicas con estos programas, como usar un procesador de texto para escribir, una hoja de cálculo para contabilidad o un programa de diseño asistido por computadora para dibujar. Las otras dos capas, los controladores de dispositivos y el sistema operativo, desempeñan importantes funciones de soporte. Su sistema puede ejecutar un programa de aplicación a la vez, o puede ejecutar varios simultáneamente.

- Controladores de dispositivos: son un conjunto de programas altamente especializados. Los controladores de dispositivo ayudan a los programas de aplicación y al sistema operativo a realizar sus tareas. Los controladores de dispositivos (en particular, los adaptadores) no interactúan con usted. Interactúan directamente con los elementos del hardware de la computadora y protegen los programas de aplicación de las especificaciones del hardware de las computadoras.

- Sistema operativo: es una colección de programas que controla la ejecución de programas y organiza los recursos de un sistema informático. Estos recursos son los componentes de hardware del sistema, como teclados, impresoras, monitores y unidades de disco. Su sistema operativo AIX viene con programas, llamados comandos o utilidades, que mantienen sus archivos, envían y reciben mensajes, brindan información diversa sobre su sistema, entre otros.

2.3. Tiempo de ejecución

El tiempo de ejecución del programa se calcula como la suma de los tiempos de ejecución de cada una de las sentencias que componen un programa. El tiempo de ejecución o tiempo de CPU, es la cantidad total de tiempo que se ejecuta el proceso; ese tiempo es generalmente independiente del tiempo de inicio, pero a menudo depende de los datos de entrada.

Además del rendimiento técnico simple, como mirar de cerca la RAM y la CPU, es útil monitorear el tiempo de ejecución de una determinada tarea. Tareas como aumentar la clasificación de un conjunto de valores pueden llevar mucho tiempo según el algoritmo utilizado.

2.3.1. Técnicas para medir el tiempo de ejecución

La puntualidad del software determina la velocidad del sistema, la premisa de esta frase es ignorar las limitaciones del hardware. Pero para la mayoría de los proyectos de software, son los recursos de hardware los que realmente limitan la velocidad de respuesta del sistema.

2.3.1.1. Utilización de un cronómetro

La forma más sencilla e intuitiva de medir el tiempo es utilizar un cronómetro manualmente desde el momento en que inicia un programa. Suponiendo que se consiga poner en marcha el cronómetro y el programa al mismo tiempo, la opción requiere que el programa imprima una determinada advertencia al final de la ejecución o al final de una subrutina.

2.3.1.2. Tiempo de utilidad

Una alternativa más rápida y automatizada al cronómetro manual es la utilidad de tiempo, que es muy fácil de usar y no requiere ajustes especiales. Es suficiente usar la siguiente sintaxis *time. /program-name* donde *program-name* es el nombre del archivo o alternativamente el comando a ejecutar.

Junto con el resultado del programa, la utilidad de tiempo imprime tres tipos diferentes de tiempo:

- Tiempo real: corresponde a la medida de tiempo real (en términos de segundos) utilizando el reloj interno del sistema.

- Tiempo usuario: corresponde a la medida del tiempo en que se ejecutan las instrucciones de la sesión “usuario”.

- Tiempo del sistema: corresponde a la medida del tiempo en el que se ejecutan las instrucciones de la sesión “supervisor”, generalmente un tiempo mucho menor que el tiempo del usuario al realizar determinadas tareas.

2.3.1.3. Contando el reloj

Para ejecutar un programa correctamente, es esencial que todas las subpartes estén sincronizadas en el momento adecuado. Físicamente, una computadora está compuesta de señales eléctricas y si se retrasan o se superponen, es posible que el sistema no responda correctamente.

El conductor responsable de sincronizar todas las partes es una señal 1 “universal” llamada *clock*. Es generado por un mineral (a menudo cuarzo) dentro de la CPU y la frecuencia de la señal (es decir, cuando va de 0 a 1) está determinada por la frecuencia de la CPU. Por ejemplo, con una CPU de 1 GHz, estamos hablando de 10 ^ 9 veces que la señal del reloj pasa de 0 a 1 en un segundo, por lo que se necesitan aproximadamente 10 ^ -9 segundos para pasar de 0 a 1.

Los componentes internos del procesador están construidos de tal manera que se "estresan" sólo en el momento en que cambia la señal del reloj. Si los dispositivos electrónicos responden en el flanco ascendente (es decir, cuando el reloj pasa de 0 a 1), hablamos de flanco ascendente y flanco descendente cuando se pasa de 1 a 0.

Siendo este el caso, es intuitivo pensar que, si tiene la cantidad de ciclos de reloj realizados en un segundo y conoce la diferencia de ciclos de reloj calculada entre dos instantes, puede convertirla en segundos.

2.4. Microservicios 4

Garzas (2015) en su artículo de internet titulado *¿Qué son los microservicios?* explica este concepto como:

> Es un concepto para construir una solución de *software* como una serie de diversas funciones configuradas para resolver un objetivo en específico, cada uno ejecutándose de forma autónoma y comunicándose entre sí, por ejemplo, a través de peticiones HTTP a sus API. Comúnmente el sistema está compuesto por un número en específico de servicios que operan entre ellos y resuelven funciones en común, algunas podrían ser; operaciones a la base de datos, pero cada microservicio es pequeño y corresponde a un área de negocio de la aplicación (p.1).

Según Humanes, Díaz, Férnandez y Yagüe (2017) en su artículo, *Sistemas Ciber-Físicos en la nube con soporte a la Variabilidad y Multitenencia,* definen la arquitectura de microservicios como:

> Un enfoque de componentes más pequeños e independientes entre sí, en comparación de una arquitectura de *software* tradicional y monolítica, en el que todo el sistema opera en una sola misma caja negra, los microservicios operan de forma independiente y son coordinados y sincronizados para finalizar diversas tareas en específico (p. 4).

> Cada uno de estos componentes, o procesos, son los microservicios. Este nuevo concepto en la implementación de aplicaciones de *software* valora la granularidad por ser liviana y la ventaja de compartir un objetivo y resolver con diferentes micro aplicaciones (Humanes *et al*., p. 3).

2.4.1. Netflix Eureka

Según Muñoz (2017), “Es un servidor basado en el protocolo REST desarrollado por Netflix, utilizado para el registro y localización de microservicios, balanceo de carga y tolerancia a fallos. La función es registrar la información de las diferentes instancias de microservicios existentes del ecosistema” (p.3).

2.4.2. Registrador de servicios

El autor Mouat (2017), en su libro *Using Docker* explica este concepto como:

> Es un servidor para el registro de los diferentes microservicios. Se encarga principalmente de registrar los microservicios de la aplicación, de conocer su localización y su estado, para saber qué hacer en cada caso. Básicamente es el cerebro de una arquitectura debido a que es el que debe saber el estado y localización de cada microservicio (p.223).

2.5. Virtualización

La virtualización, es crear un ambiente basado en *software* o *hardware*, para simular la existencia de un dispositivo físico. Estos pueden ser sistemas operativos, equipo de cómputo, dispositivos de almacenamiento, entre otros. Durante la virtualización se tiene la ventaja de crear diferentes dispositivos en el menor tiempo posible, permite copiar características de *hardware* y *software* de otros ya existentes, cada máquina virtual trabaja de forma independiente y ejecuta operaciones diversas mientras comparten recursos físicos del equipo de *hardware* que interactúa como host (López, 2017).

Cada máquina virtual tiene la característica de interactuar de forma independiente con otros dispositivos, aplicaciones y usuarios, como si se tratara de un recurso físico independiente (López, 2017).

Una máquina virtual puede admitir procesos individuales o un sistema completo según el nivel de abstracción donde se produce la virtualización. Algunas permiten el uso de *hardware* flexible y el aislamiento de *software*, mientras que otras se traducen de un conjunto de instrucciones de programación (López, 2017).

2.6. Contenedor de *software*

Ling-Hong (2016), expresa que: “Los contenedores se diferencian de las máquinas virtuales tradicionales en que los recursos del sistema operativo y no del *hardware* se comparten de forma transparente (virtualizada). Varios contenedores comparten un único núcleo de sistema operativo, lo que ahorra recursos considerables” (p.2).

Los Contenedores de *Software*, proporcionan los recursos necesarios para que las aplicaciones, variables, servicios, librerías, entre otros, puedan operar correctamente, un contenedor de *software* puede ser comparado con una máquina virtual, con la diferencia de que estos contenedores deben tener la característica de ser livianos y ser transportables, como su nombre lo indica, deben de contener las aplicaciones que ahí operan en conjunto (Zhang, 2018).

2.7. *Kubernetes*

Ruelas (2017) en su trabajo de graduación, *Modelo de Composición de Microservicios para la Implementación de una Aplicación Web de Comercio Electrónico utilizando Kubernetes,* describe el concepto de *Kubernetes* como:

> Un sistema de orquestación de contenedores, un contenedor es como una máquina virtual ligera, es el plano de control de la arquitectura, quien es el responsable de la planificación de despliegues, tiene el funcionamiento de puerta de enlace para la API y para la administración general del clúster que es un conjunto de máquinas virtuales o servidores físicos (p.34).

2.7.1. Docker

Según Ruelas (2017), “Es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones en contenedores de *software*, proporcionan una capa de abstracción y automatización de virtualización a nivel de sistema operativo en Linux” (p.35).

2.7.2. Docker Swarm

El autor Guangaun (2021) expone en su trabajo de investigación, *Uso del Algoritmo Ant Colony en la Estrategia de Programación Basada en la Plataforma en la nube Docker* define este concepto como:

> Docker Swarm es una herramienta de orquestación que se utiliza para la gestión de clústeres de contenedores para desarrolladores. El uso de Docker Swarm para la administración de contenedores puede hacer que los usuarios se sientan como si estuvieran en un solo contenedor (p.11).

De hecho, este contenedor único es el resultado del trabajo colaborativo del clúster de contenedores en segundo plano. Administra las herramientas para la abstracción de recursos de los hosts físicos.

2.7.3. Sistema de monitoreo

Según Cols (2017), “Monitorear el comportamiento de los microservicios es sumamente importante debido a que ayuda a prevenir comportamientos no deseados y corregirlos rápidamente, la complejidad operacional también se incrementa debido a las mayores demandas en la gestión de cada microservicio y el monitoreo” (p.20).

Rivera (2018), describe que: “Se trata de proporcionar una gran versatilidad para consultar prácticamente cualquier parámetro de interés de un sistema y generar alertas, que puedan ser recibidas por los responsables correspondientes mediante correo electrónico o mensajes SMS” (p.3).

2.7.4. Comunicación entre microservicios

Los autores Gómez, Anaya y Cano, (2017), en su trabajo de investigación, *Un Acercamiento a los Microservicios* plantean este concepto de la siguiente manera:

La principal diferencia entre un enfoque SOA y los microservicios es a nivel de granularidad de los microservicios es muy fina, mientras que en un enfoque SOA se configura toda la funcionalidad de los servicios de una forma enfocada en el negocio y con la utilización protocolos de comunicación, la estructura interna que compone una arquitectura basada en microservicios tiene una característica de granularidad fina.

Cada uno de ellos especializados y programados para resolver un objetivo en específico y con el uso de protocolos de comunicación estándar como HTTP mediante APIS RESTFUL (p.3).

2.7.5. *Chronograf*

El investigador Arias (2016), autor del trabajo de investigación, *Demostrador IOT-Cloud en Tiempo Real* define:

> Es una herramienta desarrollada por *InfluxData* con fines de monitorización con una configuración en la base de datos llamada *InfluxDB*. Permite una integración con esta base de datos muy fácil de configurar, simplemente se conecta el origen de la fuente de la base de datos y se configuran las herramientas de control con la generación de querys o consultas la base de datos *InfluxDB*, además permite el uso de plantillas de configuración fácil y rápido (p.48).

Esta base de datos se utilizará para almacenar cada uno de los eventos o errores que se registren dentro de un sistema basado en microservicios, depende del historial y cantidad de registros en la base de datos, éstos pueden proporcionar información suficiente para permitir al programa de monitoreo tome decisiones para solucionar el problema y así presentar una solución a un problema en un contendor y evitar se degrade completamente sistema.

2.8. *Open Source*

Es un término que ha ganado popularidad recientemente como una forma de describir la tradición de estándares abiertos, código fuente compartido y desarrollo colaborativo detrás de *software* como los sistemas operativos Linux y FreeBSD, el servidor web Apache, Perl, Tcl y Los lenguajes Python y gran parte de la infraestructura de Internet y muchos otros programas.

2.8.1. Características

Con el ascenso de Microsoft al dominio en la industria del *software*, es fácil pensar en el *software* principalmente como un producto, algo que se desarrolla, empaqueta y vende. De hecho, las aplicaciones de PC comprimidas representan solo una pequeña fracción del *software* total en uso.

La mayoría de las aplicaciones comerciales grandes se desarrollan internamente o están tan personalizadas que bien podrían serlo. La mayoría de las aplicaciones científicas son "únicas" o, si se construyen sobre herramientas estándar, incluyen un componente personalizado de gran tamaño. La administración, ya sea de una red, un gran sistema informático o un sitio web, requiere el desarrollo constante de pequeñas herramientas, scripts y "aplicaciones de cola" para que todo funcione en conjunto. Incluso en el entorno de productividad de escritorio, los usuarios avanzados desarrollan macros y otros "programas" para automatizar tareas repetitivas.

2.9. Microsoft Azure

Microsoft Azure es un servicio de computación en la nube. Azure ofrece una variedad de opciones de *software* como servicio (SaaS), plataforma como servicio (PaaS) e infraestructura como servicio (IaaS) para implementar aplicaciones y servicios en la infraestructura del centro de datos administrado por Microsoft. Microsoft Azure admite cualquier herramienta, lenguaje o marco: Node.js, Java, .NET y más. Las mejores herramientas de desarrollo de su clase de Microsoft ayudan a genera código agilizando su escritura.

Microsoft Azure se ejecuta, tanto en PC, como en Mac. Azure, puede admitir aplicaciones tan grandes y complejas dependiendo de las características los recursos configurados. La integración y la entrega continuas (CI / CD) acortarán sus ciclos de desarrollo. Permite mover los entornos de prueba a la nube para aprovisionar, poner en marcha y derribar entornos en un instante.

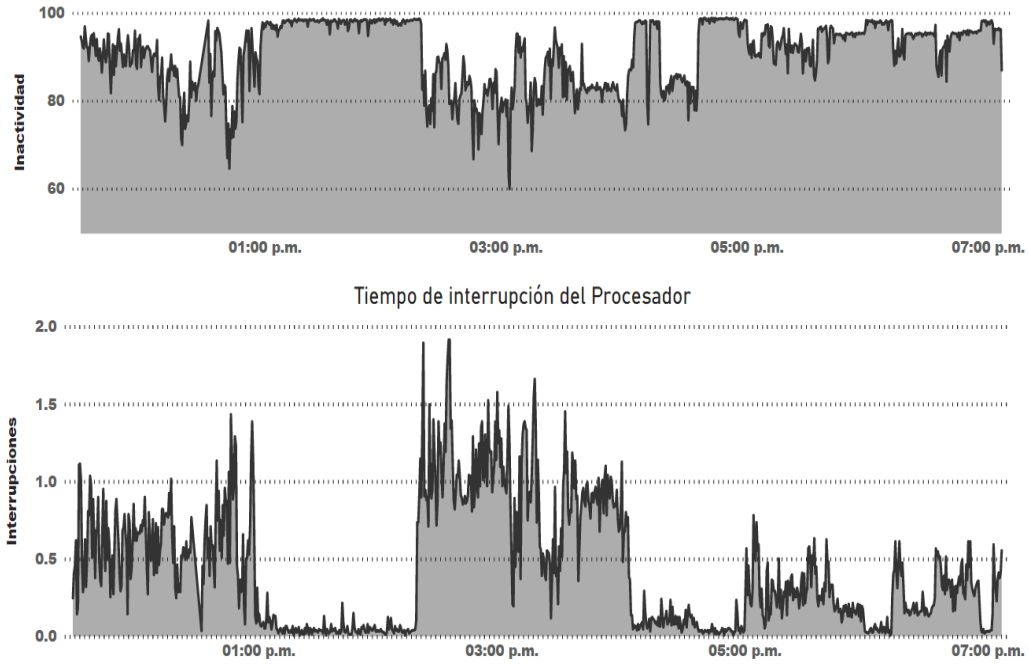
3. PRESENTACIÓN DE RESULTADOS

De acuerdo con los objetivos propuestos se presentan los siguientes resultados y gráficos utilizados para resolver cada uno de los objetivos planteados para este trabajo de investigación. Se resolvieron primero los objetivos secundarios y luego por último el objetivo principal.

El sistema en el cual se aplicó el estudio operaba en el horario de 8:00 AM hasta las 6:00 PM de lunes a viernes, durante este tiempo, se estuvo registrando valores de comportamiento en el sistema para identificar cuando el sistema se encuentra en bajo rendimiento, posibles soluciones que se tuvieron que aplicar para mejorar el rendimiento del sistema.

Se implementaron 5 microservicios, los cuales se encontraban en constante comunicación vía *html*, enviando y recibiendo mensajes, durante esa comunicación, debido a que la información que se capturó es bastante, una de las mejores herramientas para trabajar con grandes volúmenes de datos se consideró el uso de *PowerBI*, con este programa se generaron las siguientes gráficas.

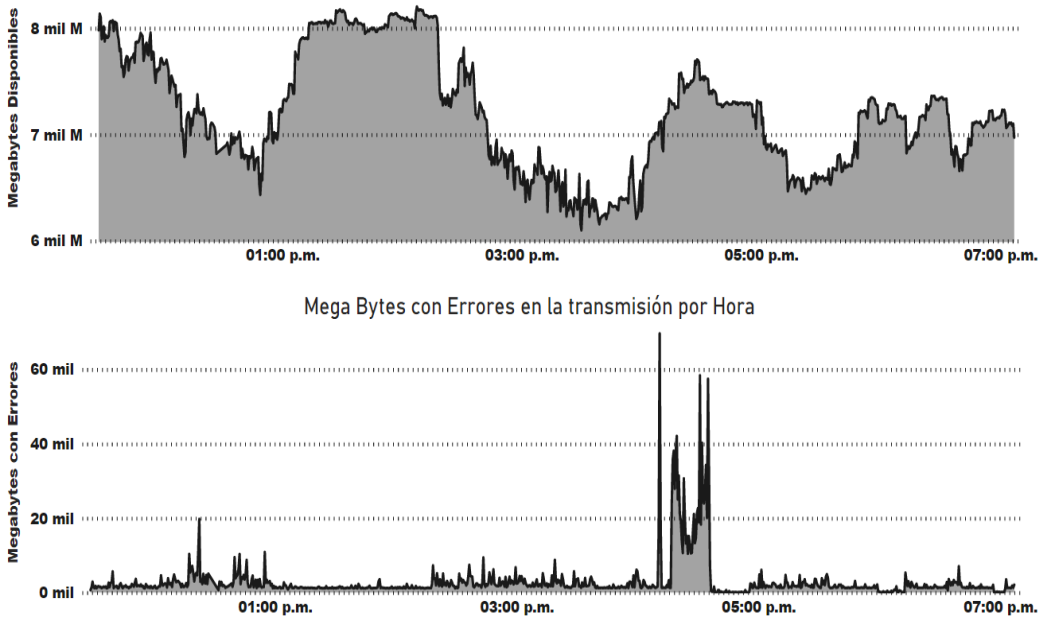
Figura 1. **Tiempo de inactivo vs. interrupción en el procesador de un microservicio**



Fuente: elaboración propia, empleando *Power BI.*

Se puede apreciar que aproximadamente desde la 1:00 PM hasta las 2:20 PM se tuvo inactividad completamente casi 0, esto se puede entender a que el sistema no opero, por lo tanto, se puede considerar que el sistema dejó de operar.

Figura 2. **Disponibilidad de memoria, medida en megabytes**



Fuente: elaboración propia, empleando *Power BI*.

Se puede observar la disponibilidad de memoria que existe en un día de operaciones normal, desde las 8:00 AM hasta las 7:00 PM, durante los periodos de 3:00 PM hasta las 5:00 PM fue cuando el sistema exigió más memoria y como resultado aumentaron los errores de transición.

3.1. Objetivo 1

Este objetivo consiste en establecer la configuración e interconexión de una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala.

3.1.1. Análisis descriptivo de la información

A diferencia de una aplicación monolítica en la que todo se ejecuta dentro de una sola instancia, una aplicación nativa de la nube consiste en servicios independientes distribuidos en máquinas virtuales, contenedores y regiones geográficas. Administrar las opciones de configuración para docenas de servicios interdependientes puede ser un desafío. Las copias duplicadas de los ajustes de configuración en diferentes ubicaciones son propensas a errores y difíciles de administrar. La configuración centralizada es un requisito crítico para las aplicaciones nativas de la nube distribuidas.

3.1.2. Creación de una instancia de Azure Spring Cloud

Se creó una instancia Azure Spring Cloud con la CLI de Azure. Es posible realizar exactamente la misma configuración mediante Azure Portal.

El nombre debe ser único entre todas las instancias de Azure Spring Cloud en todo Azure. El nombre solo puede contener letras minúsculas, números y guiones. El primer carácter debe ser una letra. El último carácter debe ser una letra o un número. El valor debe tener entre 4 y 32 caracteres. Para la creación del Spring Cloud se utilizó la siguiente instrucción por consola.

```
RESOURCE_GROUP_NAME=spring-cloud-workshop
SPRING_CLOUD_NAME=azure-spring-cloud-workshop
```

Con estas variables establecidas, se procede a la creación un grupo de recursos. En el siguiente script, el grupo de recursos se encuentra en la región (mediante el argumento), pero se puede elegir una región más cercana para obtener un mejor rendimiento, por ejemplo: eastus-l eastusaz account list-locations

```
az group create \
    -g "$RESOURCE_GROUP_NAME" \
    -l eastus

az spring-cloud create \
    -g "$RESOURCE_GROUP_NAME" \
    -n "$SPRING_CLOUD_NAME" \
    --sku standard \
    --enable-java-agent
```

3.1.3. Crear un repositorio de Git para almacenar la configuración de la aplicación

Es necesario la creación de una cuenta GitHub, se debe de configurar el repositorio de una forma privada donde se almacenarán las configuraciones del Spring Boot.

En el nuevo repositorio privado de GitHub, contiene un nuevo archivo, que almacena los datos de configuración de todos los microservicios, el archivo puede llevar el nombre de *microservicios.application.yml*

Normalmente, cada aplicación *spring boot* incluye un archivo de este tipo dentro de los binarios de la aplicación para contener la configuración de la aplicación. Un servidor de configuración de Spring Cloud permite que dicha configuración se almacene fuera de la aplicación, lo que proporciona las siguientes ventajas:

- Permite almacenar parámetros confidenciales (como la contraseña de su base de datos) fuera de su aplicación.

- Su configuración se almacena en un repositorio de Git, por lo que sus datos se pueden etiquetar o revertir.

- Utiliza un repositorio Git específico, que se puede proteger por separado.

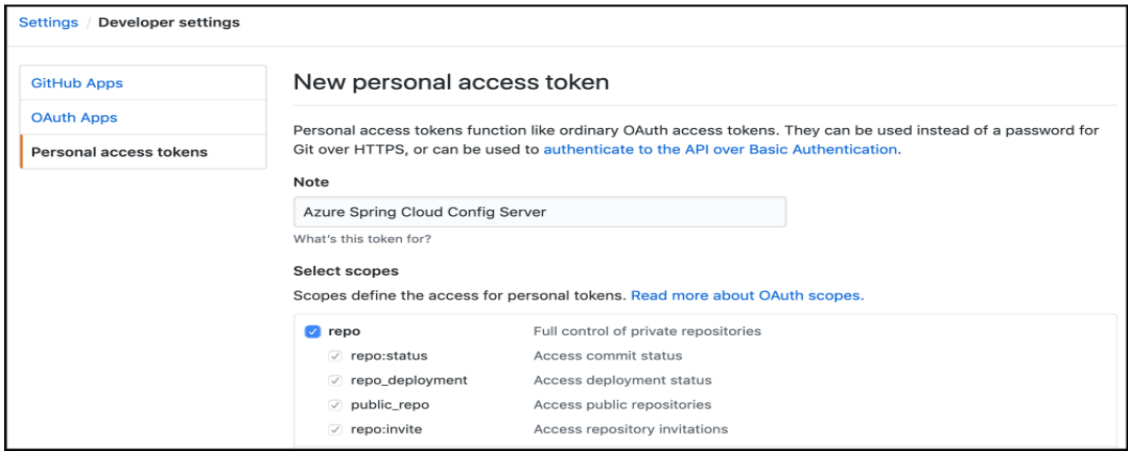
- Proporciona un lugar centralizado para almacenar todos sus datos de configuración, para todos sus microservicios.

3.1.4. Crear un *token* personal de GitHub

Azure Spring Cloud puede acceder a repositorios de Git que son públicos, protegidos por SSH o protegidos mediante la autenticación básica HTTP. Se usó esta última opción, ya que es más fácil de crear y administrar con GitHub.

Es necesario la creación de un *token* personal. Cuando el sistema solicita esta información es necesario seleccionar los ámbitos, marcando toda la sección como "repositorio" como se muestra a imagen.

Figura 3. **Generación de nuevo *token* de acceso personal**



Fuente: Microsoft Azure (2021) *Portal de Microsoft Azure.*

3.1.5. Crear un microservicio spring boot

En este microservicio se usó Spring Data JPA para leer y escribir datos de una base de datos de Azure para MySQL:

Esa base de datos se vinculará automáticamente a nuestro servicio mediante Azure *Spring Cloud*. Azure *Database for MySQL* es una versión totalmente administrada de *MySQL* que se ejecuta en Azure.

Para crear la aplicación se ejecutó la siguiente instrucción:

az spring-cloud app create --name todo-service --resource-group "$RESOURCE_GROUP_NAME" --service "$SPRING_CLOUD_NAME"

En el siguiente paso es necesario asociar una base de datos Azure para *MySql* y asociar al spring-cloud

```
az mysql server create \
    --name ${SPRING_CLOUD_NAME}-mysql \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --sku-name B_Gen5_1 \
    --storage-size 5120 \
    --admin-user "spring"
```

En este punto fue necesario la creación de una base de datos en la cual cada vez que se tenga que configurar un nuevo microservicio, se pueda conectar a ésta, para ello se ejecutó la siguiente instrucción:

```
az mysql db create \
    --name "todos" \
    --server-name ${SPRING_CLOUD_NAME}-mysql
```

```
az mysql server firewall-rule create \
    --name ${SPRING_CLOUD_NAME}-mysql-allow-azure-ip \
    --resource-group "$RESOURCE_GROUP_NAME" \
    --server ${SPRING_CLOUD_NAME}-mysql \
    --start-ip-address "0.0.0.0" \
    --end-ip-address "0.0.0.0"
```

3.1.6. Implementar la aplicación

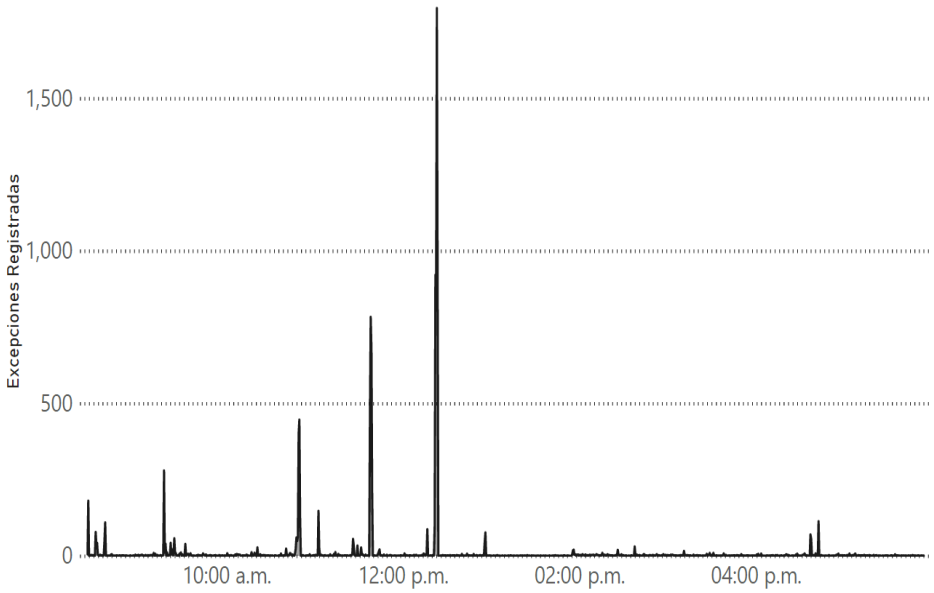
Una vez configurado todo lo anterior, el microservicio ya se encuentra listo para su implementación, fue necesario ejecutar las siguientes instrucciones:

```
cd todo-service
./mvnw clean package -DskipTests
az spring-cloud app deploy --name todo-service --service "$SPRING_CLOUD_NAME" --resource-group "$RESOURCE_GROUP_NAME" --jar-path target/demo-0.0.1-SNAPSHOT.jar
```

3.1.7. Análisis de variables

En la siguiente gráfica, se presentan los resultados en la mejora en la conexión entre los diferentes microservicios del sistema, analizando las excepciones que el sistema generó después de las configuraciones anteriormente descritas.

Figura 4. **Presentación de excepciones por hora**



Fuente: elaboración propia, empleando *Power BI*.

Se puede observar que después de la 1:00 PM aproximadamente el sistema se estabilizó totalmente, al ya no generar excepciones fuertes que puedan representar la caída total del sistema.

3.2. Objetivo 2

En este objetivo se determina cómo se asegura la comunicación entre microservicios ante cualquier ataque de una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala

La comunicación entre microservicios debe ser eficiente y robusta. Con muchos servicios pequeños que interactúan para completar una sola actividad comercial, esto puede ser un desafío.

Estos son algunos de los principales desafíos que surgen de la comunicación de servicio a servicio. Las mallas de servicio, que se describen más adelante en esta investigación, están diseñadas para manejar muchos de estos desafíos.

Puede haber docenas o incluso cientos de instancias de cualquier microservicio dado. Una instancia puede fallar por cualquier número de razones. Puede haber un error a nivel de nodo, como un error de *hardware* o un reinicio de máquina virtual. Una instancia puede bloquearse o verse abrumada por las solicitudes y no poder procesar ninguna solicitud nueva. Cualquiera de estos eventos puede provocar un error en una llamada de red. Hay tres patrones de diseño que pueden ayudar a que las llamadas de red de servicio a servicio sean más resistentes:

3.2.1. Vuelva a intentarlo

Una llamada de red puede fallar debido a una falla transitoria que desaparece por sí sola. En lugar de fallar directamente, la persona que llama normalmente debe volver a intentar la operación un cierto número de veces, o hasta que transcurra un período de tiempo de espera configurado. Sin embargo, si una operación no es idempotente, los reintentos pueden causar efectos secundarios no deseados. La llamada original puede tener éxito, pero la persona que llama nunca recibe una respuesta. Si el autor de la llamada reintenta, la operación se puede invocar dos veces. En general, no es seguro volver a intentar los métodos POST o PATCH, porque no se garantiza que sean idempotentes.

3.2.2. Disyuntor

Demasiadas solicitudes fallidas pueden causar un cuello de botella, ya que las solicitudes pendientes se acumulan en la cola. Estas solicitudes bloqueadas pueden contener recursos críticos del sistema, como memoria, subprocesos, conexiones de bases de datos, entre otros, lo que puede provocar errores en cascada. El patrón de disyuntor puede evitar que un servicio intente repetidamente una operación que es probable que falle.

3.2.3. Equilibrio de carga

Cuando el servicio "A" llama al servicio "B", la solicitud debe llegar a una instancia en ejecución del servicio "B". En *Kubernetes*, el tipo de recurso proporciona una dirección IP estable para un grupo de pods. El tráfico de red a la dirección IP del servicio se reenvía a un pod mediante reglas *iptable*.

De forma predeterminada, se elige un pod aleatorio. Una malla de servicio puede proporcionar algoritmos de equilibrio de carga más inteligentes basados en la latencia observada u otras métricas de servicio.

3.2.4. Mensajería sincrónica frente a asincrónica

Hay dos patrones básicos de mensajería que los microservicios pueden usar para comunicarse con otros microservicios.

Comunicación síncrona. En este patrón, un servicio llama a una API que otro servicio expone, utilizando un protocolo como HTTP o gRPC. Esta opción es un patrón de mensajería sincrónica porque la persona que llama espera una respuesta del receptor.

Paso de mensajes asincrónicos. En este patrón, un servicio envía un mensaje sin esperar una respuesta y uno o más servicios procesan el mensaje de forma asincrónica.

Es importante distinguir entre E/S (Entradas/Salida) asincrónicas y un protocolo asincrónico. La E/S asincrónica significa que el subproceso de llamada no está bloqueado mientras se completa la E/S. Eso es importante para el rendimiento, pero es un detalle de implementación en términos de la arquitectura. Un protocolo asincrónico significa que el remitente no espera una respuesta. HTTP es un protocolo sincrónico, aunque un cliente HTTP puede utilizar E/S asincrónicas cuando envía una solicitud.

Hay compensaciones para cada patrón. La solicitud/respuesta es un paradigma bien entendido, por lo que diseñar una API puede parecer más natural que diseñar un sistema de mensajería. Sin embargo, la mensajería asincrónica tiene algunas ventajas que pueden ser útiles en una arquitectura de microservicios:

- Múltiples suscriptores. Usando un modelo pub/sub, varios consumidores pueden suscribirse para recibir eventos. Consulte Estilo de arquitectura basada en eventos.

- Aislamiento de fallos. Si el consumidor falla, el remitente aún puede enviar mensajes. Los mensajes se recogerán cuando el consumidor se recupere. Esta capacidad es especialmente útil en una arquitectura de microservicios, porque cada servicio tiene su propio ciclo de vida. Un servicio podría dejarse de estar disponible o ser reemplazado por una versión más reciente en un momento dado.

La mensajería asincrónica puede manejar el tiempo de inactividad intermitente. Las API síncronas, por otro lado, requieren que el servicio descendente esté disponible o la operación falla.

- Acoplamiento reducido. El remitente del mensaje no necesita saber sobre el consumidor.

- Capacidad de respuesta. Un servicio ascendente puede responder más rápido si no espera en los servicios descendentes. Esto es especialmente útil en una arquitectura de microservicios. Si hay una cadena de dependencias de servicio (el servicio A llama a B, que llama a C, entre otros), esperar en llamadas sincrónicas puede agregar cantidades inaceptables de latencia.

- Nivelación de carga. Una cola puede actuar como un búfer para nivelar la carga de trabajo, de modo que los receptores puedan procesar los mensajes a su propio ritmo.

- Flujos de trabajo. Las colas se pueden usar para administrar un flujo de trabajo, marcando el mensaje después de cada paso del flujo de trabajo.

Sin embargo, también hay algunos desafíos para usar la mensajería asincrónica de manera efectiva.

- Acoplamiento con la infraestructura de mensajería. El uso de una infraestructura de mensajería en particular puede causar un acoplamiento estrecho con esa infraestructura. Será difícil cambiar a otra infraestructura de mensajería más adelante.

- Latencia. La latencia de extremo a extremo para una operación puede llegar a ser alta si las colas de mensajes se llenan.

- Costo. Con altos rendimientos, el costo monetario de la infraestructura de mensajería podría ser significativo.

- Complejidad. El manejo de mensajes asincrónicos no es una tarea trivial. Por ejemplo, debe controlar los mensajes duplicados, ya sea eliminando la duplicidad o haciendo que las operaciones sean idempotentes. También es difícil implementar la semántica de solicitud-respuesta utilizando mensajería asincrónica. Para enviar una respuesta, necesita otra cola, además de una forma de correlacionar los mensajes de solicitud y respuesta.

- Rendimiento. Si los mensajes requieren semántica de cola, la cola puede convertirse en un cuello de botella en el sistema. Cada mensaje requiere al menos una operación de cola y una operación de descola. Además, la semántica de cola generalmente requiere algún tipo de bloqueo dentro de la infraestructura de mensajería. Si la cola es un servicio administrado, puede haber latencia adicional, ya que la cola es externa a la red virtual del clúster. Puede mitigar estos problemas mediante mensajes por lotes, pero eso complica el código. Si los mensajes no requieren semántica de cola, es posible que pueda usar una secuencia de eventos en lugar de una cola.

3.2.5. Transacciones distribuidas

Un desafío común en los microservicios es manejar correctamente las transacciones que abarcan múltiples servicios. A menudo, en este escenario, el éxito de una transacción es todo o nada: si uno de los servicios participantes falla, toda la transacción debe fallar. Existen dos casos a considerar:

- Un servicio puede experimentar un error transitorio, como un tiempo de espera de red. Estos errores a menudo se pueden resolver simplemente reintentando la llamada. Si la operación sigue fallando después de un cierto número de intentos, se considera una falla no transitoria.

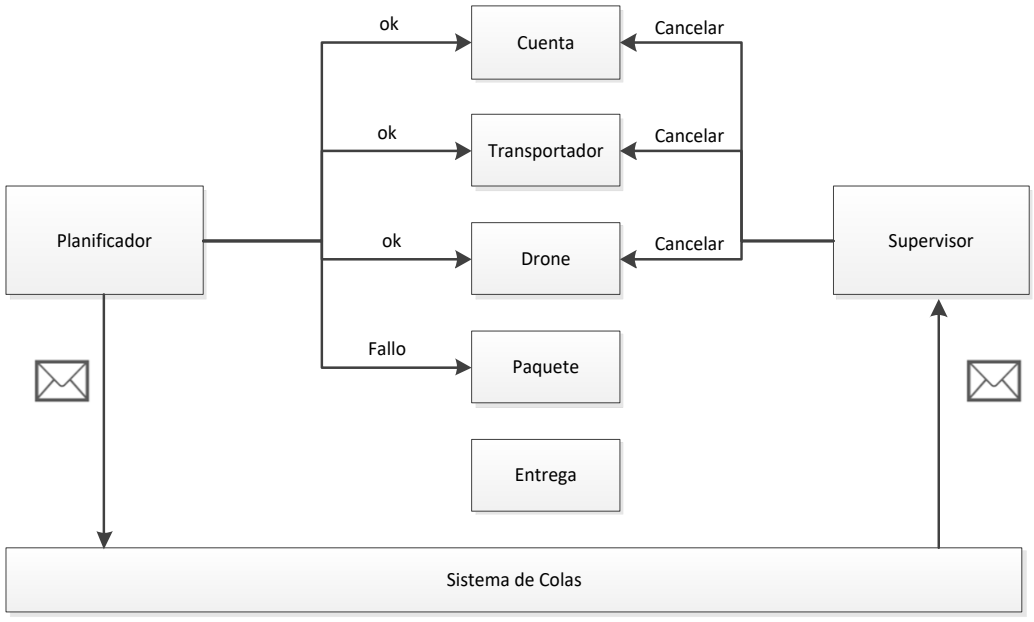
- Una falla no transitoria, es cualquier falla que es poco probable que desaparezca por sí sola. Los errores no transitorios incluyen condiciones de error normales, como la entrada no válida. También incluyen excepciones no controladas en el código de la aplicación o un proceso que se bloquea. Si se produce este tipo de error, toda la transacción comercial debe marcarse como un error. Puede ser necesario deshacer otros pasos en la misma transacción que ya se realizaron correctamente.

- Después de un error no transitorio, la transacción actual podría estar en un estado parcialmente fallido, donde uno o más pasos ya se completaron correctamente. Por ejemplo, si un microservicio ya programó una entrega y se encuentra parcialmente fallido, todo el servicio debe cancelarse. En ese caso, la aplicación debe deshacer los pasos que se realizaron correctamente, mediante una transacción de compensación. En algunos casos, esto debe hacerse mediante un sistema externo o incluso mediante un proceso manual.

- Si la lógica para compensar transacciones es compleja, es necesario considerar la posibilidad de crear un servicio independiente que sea responsable de este proceso. En la aplicación *Drone Delivery*, el servicio *Scheduler* coloca las operaciones fallidas en una cola dedicada. Un microservicio separado, llamado Supervisor, lee de esta cola y llama a una API de cancelación en los servicios que necesitan compensar.

Esta es una variación del patrón Scheduler Agent Supervisor. El servicio Supervisor también puede tomar otras medidas, como notificar al usuario por mensaje de texto o correo electrónico, o enviar una alerta a un panel de operaciones.

Figura 5. **Diagrama de bloques de un patrón de diseño distribuido de microservicios**



Fuente: Microsoft Docs. (2021) *Portal de Microsoft Azure*

El propio servicio *Scheduler* puede fallar (por ejemplo, porque un nodo se bloquea). En ese caso, una nueva instancia puede girar y hacerse cargo. Sin embargo, cualquier transacción que ya estuviera en curso debe reanudarse.

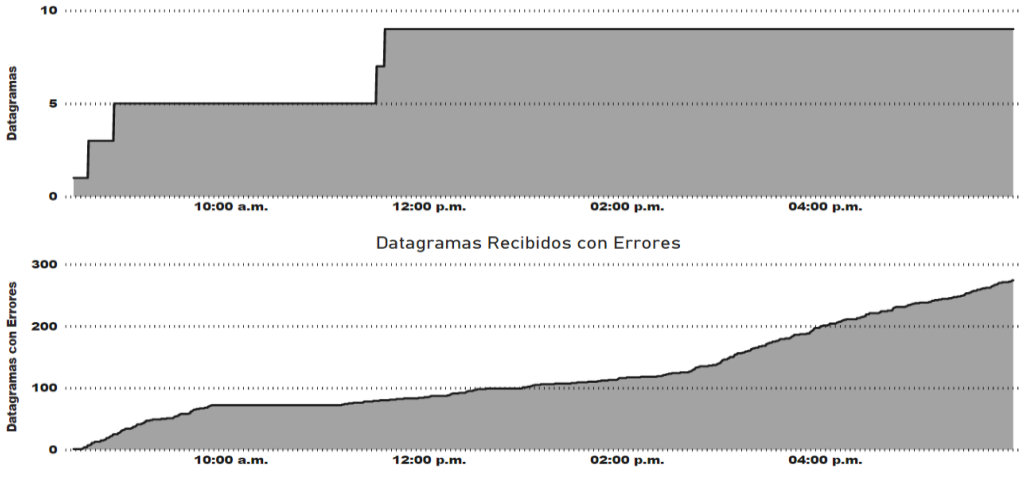
Un enfoque es guardar un punto de control en una tienda duradera después de completar cada paso en el flujo de trabajo. Si una instancia del servicio *Scheduler* se bloquea en medio de una transacción, una nueva instancia puede usar el punto de control para reanudar donde la instancia anterior lo dejó. Sin embargo, escribir puntos de control puede crear una sobrecarga de rendimiento.

Otra opción es diseñar todas las operaciones para que sean idempotentes. Una operación es idempotente si se puede llamar varias veces sin producir efectos secundarios adicionales después de la primera llamada. Esencialmente, el servicio descendente debe ignorar las llamadas duplicadas, lo que significa que el servicio debe poder detectar llamadas duplicadas. No siempre es sencillo implementar métodos idempotentes.

3.2.6. Análisis de variables

En la siguiente gráfica, se presentan los resultados en la mejora de asegurar el envío de mensajes entre el sistema basado en microservicios.

Figura 6. **Presentación de datagramas con errores**



Fuente: elaboración propia, empleando *Power BI*

En la figura se puede observar que en la gráfica *Datagramas recibidos con errores de dirección* se obtuvo una curva alcanzando una estabilidad en el valor aproximado de 300 datagramas después de observar el sistema en constante operación desde las 8 de la mañana hasta las 5 de la tarde.

3.3. Objetivo 3

Se buscó establecer cómo se garantiza la atomicidad de un componente de microservicio ante eventos de caída en una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala

Una transacción es una sola unidad de lógica o trabajo, a veces compuesta de múltiples operaciones. Dentro de una transacción, un evento es un cambio de estado que se produce en una entidad y un comando encapsula toda la información necesaria para realizar una acción o desencadenar un evento posterior.

Las transacciones deben ser atómicas, consistentes, aisladas y duraderas (ACID). Las transacciones dentro de un solo servicio son ACID, pero la consistencia de los datos entre servicios requiere una estrategia de gestión de transacciones entre servicios.

Durante la implementación de un patrón de diseño basado en microservicios, a través del cual se debe garantizar la atomicidad de la información que operan fue necesario considerar lo siguiente:

- La atomicidad es un conjunto indivisible e irreductible de operaciones que deben ocurrir o no ocurrir ninguna.

- Consistencia significa que la transacción lleva los datos solo de un estado válido a otro estado válido.

- El aislamiento garantiza que las transacciones simultáneas produzcan el mismo estado de datos que las transacciones ejecutadas secuencialmente habrían producido.

- La durabilidad garantiza que las transacciones comprometidas permanezcan comprometidas incluso en caso de falla del sistema o corte de energía.

Un modelo de base de datos por microservicio proporciona muchas ventajas para las arquitecturas de microservicios. La encapsulación de datos de dominio permite que cada servicio use su mejor tipo y esquema de almacén de datos, escale su propio almacén de datos según sea necesario y esté aislado de los errores de otros servicios. Sin embargo, garantizar la coherencia de los datos en las bases de datos específicas del servicio plantea desafíos.

Las transacciones distribuidas como el protocolo de confirmación de dos fases (2PC) requieren que todos los participantes en una transacción se confirmen o reviertan antes de que la transacción pueda continuar. Sin embargo, algunas implementaciones participantes, como las bases de datos NoSQL y la intermediación de mensajes, no admiten este modelo.

Otra limitación de las transacciones distribuidas es la sincronicidad y disponibilidad de la comunicación entre procesos (IPC). La CIP proporcionada por el sistema operativo permite que procesos separados compartan datos.

Para que las transacciones distribuidas se confirmen, todos los servicios participantes deben estar disponibles, lo que podría reducir la disponibilidad general del sistema. Las implementaciones arquitectónicas con IPC o limitaciones de transacción son candidatas para el patrón de diseño saga.

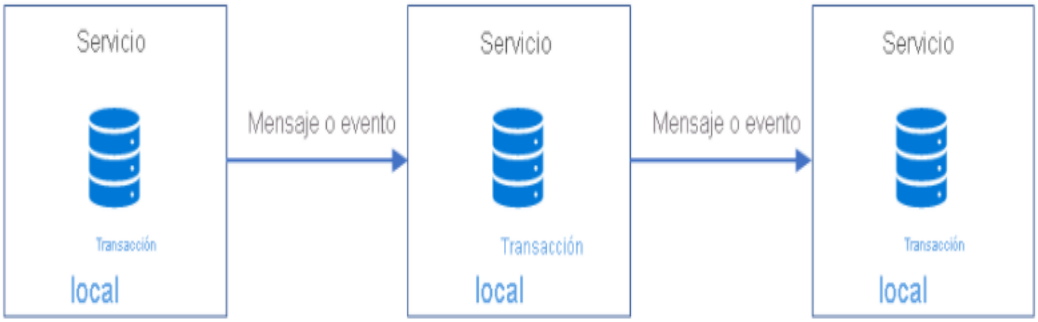
3.3.1. Patrón de diseño SAGA

Es una forma de administrar la consistencia de los datos en microservicios en escenarios de transacciones distribuidas. SAGA es una secuencia de transacciones que actualiza cada servicio y publica un mensaje o evento para desencadenar el siguiente paso de la transacción. Si un paso falla, SAGA ejecuta transacciones compensatorias que contrarrestan las transacciones anteriores.

El patrón SAGA proporciona gestión de transacciones utilizando una secuencia de transacciones locales. Una transacción local es el esfuerzo de trabajo atómico realizado por un participante del sistema. Cada transacción local actualiza la base de datos y publica un mensaje o evento para desencadenar la siguiente transacción local.

Si una transacción local falla, el patrón ejecuta una serie de transacciones compensatorias que deshacen los cambios que se realizaron por las transacciones locales anteriores.

Figura 7. **Envío de mensajes entre diferentes microservicios, utilizando el patrón de diseño SAGA**



Fuente: Microsoft Docs. (2021), *Portal de Microsoft Azure*

En este tipo de patrón es necesario considerar lo siguiente:

- Las transacciones resarcibles son transacciones que potencialmente pueden revertirse procesando otra transacción con el efecto contrario.

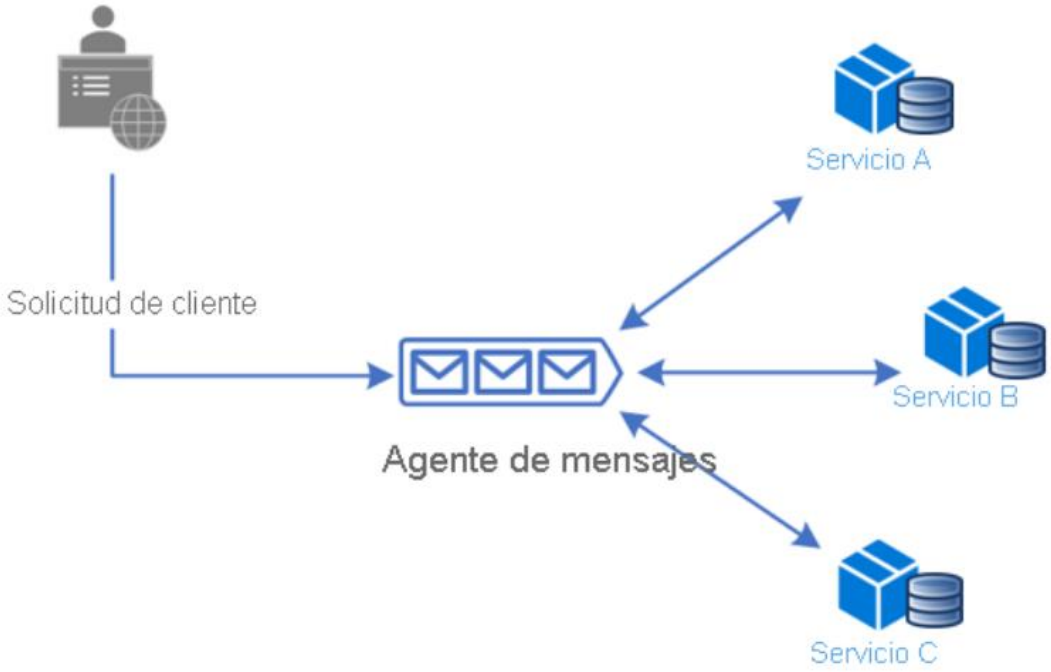
- Una transacción pivote es el punto *go/no-go*. Si la transacción dinámica se confirma, SAGA se extiende hasta su finalización. Una transacción pivote puede ser una transacción que no es ni resarcible ni recuperable, o puede ser la última transacción resarcible o la primera transacción recuperable en este patrón.

- Las transacciones recuperables son transacciones que siguen a la transacción dinámica y se garantiza que tendrán éxito.

3.3.2. Coreografía

Es una forma de coordinar sagas donde los participantes intercambian eventos sin un punto de control centralizado. Con la coreografía, cada transacción local publica eventos de dominio que desencadenan transacciones locales en otros servicios.

Figura 8. **Coreografía en la distribución de mensajes entre diferentes microservicios**



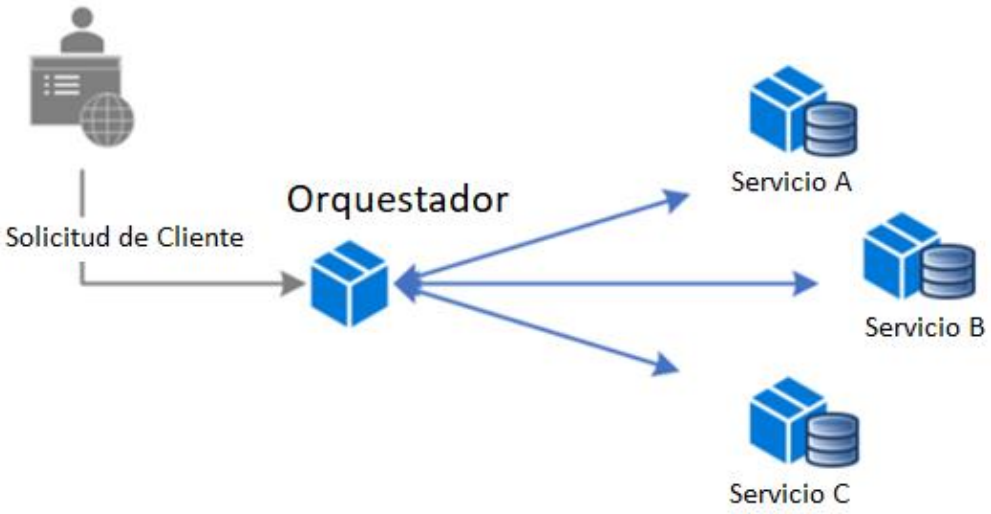
Fuente: Microsoft Docs. (2021), *Portal de Microsoft Azure*

3.3.3. Orquestación

La orquestación es una forma de coordinar el patrón de diseño SAGA donde un controlador centralizado le dice a los participantes qué transacciones locales ejecutar. El orquestador maneja todas las transacciones y les dice a los participantes qué operación realizar en función de los eventos.

El orquestador ejecuta solicitudes, almacena e interpreta los estados de cada tarea y maneja la recuperación de fallas con transacciones de compensación.

Figura 9. **Orquestador en la distribución de mensajes entre diferentes microservicios**



Fuente: Microsoft Docs. (2021), *Portal de Microsoft Azure*

3.3.4. ¿Cuándo utilizar el patrón de diseño SAGA?

Este tipo de patrón de diseño se aconseja utilizar para los casos en los que:

- Garantizar la consistencia de los datos en un sistema distribuido sin acoplamientos estrechos.
- Revertir o compensar si falla una de las operaciones de la secuencia.

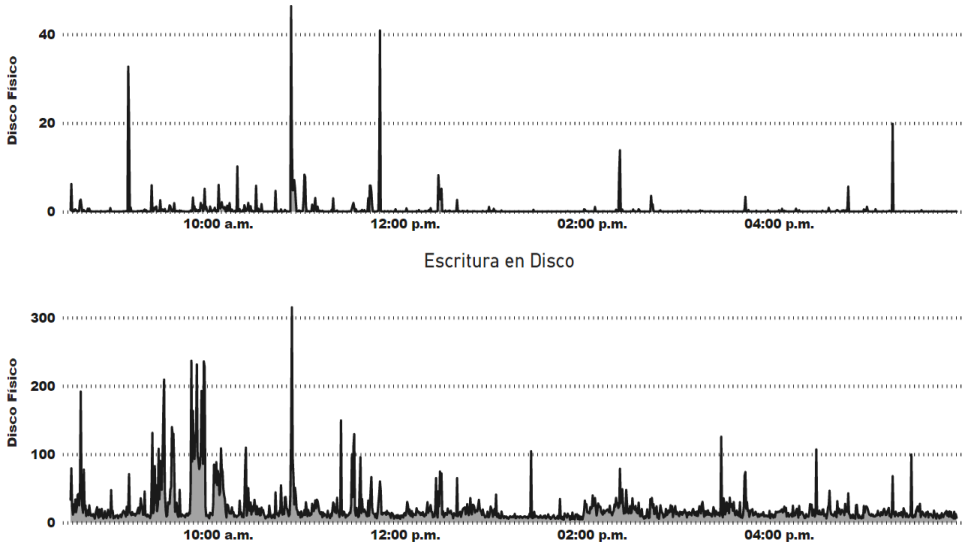
Este tipo de patrón de diseño no se aconseja utilizar para cuando:

- Las transacciones son muy estrechamente acopladas.
- Tratar de compensar las transacciones que ocurren en participantes anteriores.

3.3.5. Análisis de variables

En la siguiente gráfica, se presentan los resultados en la mejora de asegurar la atomicidad de la información que procesa un microservicio.

Figura 10. **Análisis entre el porcentaje de lectura y escritura en disco**



Fuente: elaboración propia, empleando *Power BI*

Al aplicar el patrón de diseño SAGA, se puede observar la mejora en la escritura y lectura a disco contra la cantidad de tiempo que le tomo escribir una cantidad de *bytes*, la tasa de error de escritura disminuye mejorando el porcentaje de éxito en la lectura y escritura.

3.4. Objetivo general

Evidenciar una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala

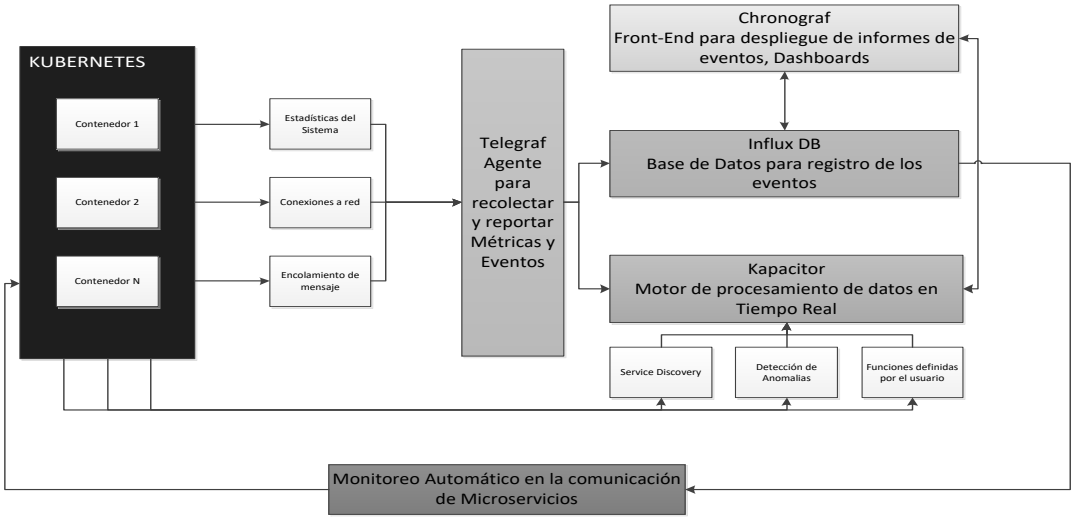
Dentro de los orquestadores, se encuentran *Dockers Swarm* o *Kubernetes*, cuando un microservicio baja en rendimiento o se apaga completamente, estos sistemas, tratan de reiniciar y levantar de forma completa el servicio nuevamente.

Como limitante a este tipo de arquitectura de *software*, no se analiza la razón por la cual estos microservicios dejaron de funcionar o tal vez fue un ataque de terceros por vulnerabilidad en la seguridad del sistema.

Dentro del marco tecnológico de este tipo de arquitectura de *software*, exponer los problemas e inconvenientes planteados en una implementación donde la comunicación entre los microservicios, no solo es distribuida por la propia naturaleza de la arquitectura, sino que además también es asíncrona, desde este punto de partida, se busca analizar, desarrollar e implementar una solución de *software* en la que se pueda integrar de una manera sencilla con cualquier arquitectura de microservicios un modelo de comunicación que resuelva eficientemente los problemas expuestos.

Se planteó la siguiente arquitectura basada en distintas soluciones de *software*, en donde la comunicación entre los componentes se configuró para eficientar la respuesta de los diferentes microservicios que componen el sistema.

Figura 11. **Definición de arquitectura al monitoreo automático en la comunicación de microservicios**



Fuente: elaboración propia, empleando *Microsoft Visio*

La arquitectura definida en la imagen de arriba describe como cada uno de los elementos se interconectan para dar solución a la problemática de monitoreo en los sistemas basados en microservicios:

Telegraf: este elemento se conectó a cada uno de los microservicios y mantuvo la monitorización constante para recolectar información a diferentes eventos como pueden ser: conexiones a red, encolamiento de mensajes, tiempos de respuesta, integridad en los mensajes.

InfluxDB: en esta base de datos, recolectó cada uno de los eventos que cada agente detecte por lapsos de tiempos, esto puede configurarse por ejemplo cada 5 minutos.

Kapacitor: se configuró para el procesado de información en *streams* o *batches*, crear alertas y detectar anomalías, realizando acciones específicas en función de estas alertas.

Hronograf: con esta interfaz de usuario, se muestran los datos guardados en la base de datos *InfluxDB*, de una forma tabulada en tiempo real.

Kubernetes: se tomó la decisión de usar este administrador de contenedores, dada la gran variedad de información que existe para la configuración entre contenedores.

Monitoreo automático: este pequeño modulo se desarrolló utilizando un *framework Visual Studio* .Net 2019, el cual se conectó directamente a la base de datos *InfluxDB*. La idea principal es que el programa se encuentra leyendo constantemente la información de la base de datos y construye una biblioteca de

eventos o anomalías registradas en cada uno de los contenedores que pueden componer un sistema de microservicios.

Conforme la base de datos fue madurando, este módulo llegó a crecer y cambiar su configuración en las directrices, para que respondan a cada uno de los eventos y con ello interactuar directamente con los sistemas orquestadores, que pueden ser *Docker Swarm* o *Kubernetes*, para ir garantizando poco a poco la atomicidad en la información que se maneja en los microservicios y que no se degrade completamente el funcionamiento de cada uno de ellos.

3.4.1. Configuración de archivo docker-compose.yml

Es necesario comenzar con la configuración del archivo Docker-compose.yml, en este caso se configura el *InfluxDB*, que es donde se va a tener almacenado todo lo que *Telegraf* recolectó e inyectó a la base de datos. *Chronograf*, es el portal o el *dashboard*, donde se va a mostrar la información de la base de datos.

```
version: "3.3"
services:
  influxdb:
    container_name: influxdb
    image: influxdb:latest
    volumes:
      - ./influxdb:/var/lib/influxdb
    ports:
      - 8086:8086
    networks:
      - "monitoring"

  chronograf:
    container_name: chronograf
    image: chronograf:latest
    ports:
      - 8888:8888
    volumes:
```

```
    - "./chronograf:/var/lib/chronograf"
  networks:
    - "monitoring"


 telegraf:
  container_name: telegraf
  image: telegraf:latest
  volumes:
    - "./telegraf.conf:/etc/telegraf/telegraf.conf:ro"
    - "/var/run/docker.sock:/var/run/docker.sock"
  networks:
    - "monitoring"

 kapacitor:
  container_name: kapacitor
  image: kapacitor:latest
  environment:
    - KAPACITOR_INFLUXDB_0_URLS_0=http://influxdb:8086
    - KAPACITOR_HOSTNAME=kapacitor
    - KAPACITOR_LOGGING_LEVEL=INFO
    - KAPACITOR_REPORTING_ENABLED=false
  volumes:
    - "./kapacitor:/var/lib/kapacitor"
  ports:
    - 9092:9092
  networks:
    - "monitoring"


volumes:
  chronograf:
  influxdb:


networks:
  monitoring:
```

También se observa que la configuración de los puertos de todos los contenedores, son expuestos, excepto el puerto que se utiliza para *Telegraf*. Se configuró un volumen de dispositivo de almacenamiento de información apuntando al archivo de configuración, el cual fue necesario para definir los “inputs”. También se observa volúmenes relacionados a *InfluxDB*,

imprescindibles para que la información sea persistente, lo mismo en la configuración de *Chronograf* y para *Kapacitor*.

Una vez definida la configuración anterior, se procedió a levantar el servicio del archivo yml, utilizando la siguiente instrucción:

```
docker-compose up -d
Creating telegraf   ... done
Creating kapacitor  ... done
Creating influxdb   ... done
Creating chronograf ... done
```

Una vez creados y configurados los contenedores, para levantar cada uno de ellos se aplicó la siguiente instrucción:

```
Docker ps
```

La terminal debería de devolver algo parecido a lo siguiente:

Figura 12. **Listado de contenedores configurados y ejecutando actualmente**

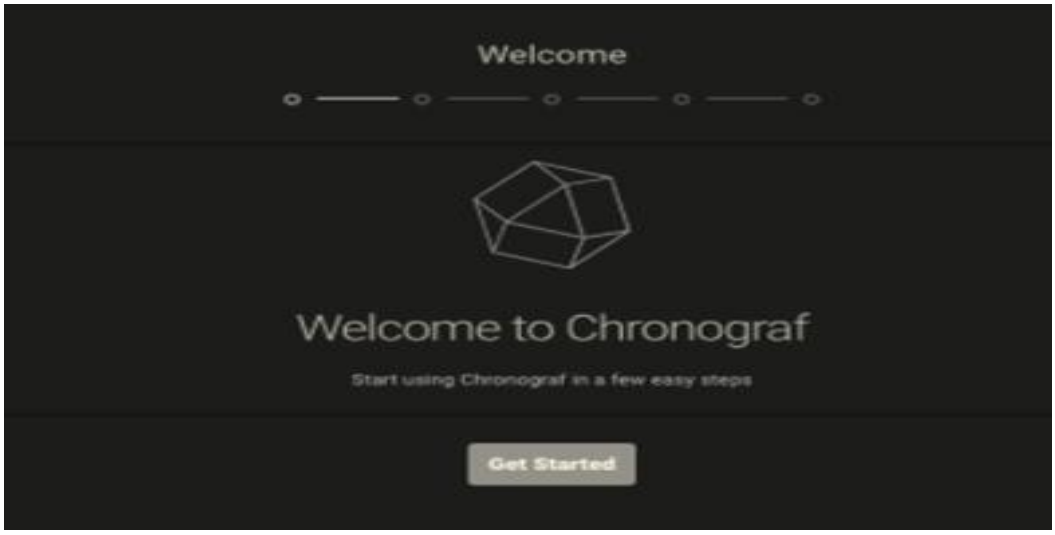
```
CONTAINER ID        IMAGE               COMMAND                  CREATED
2fa48cec540e        chronograf:latest   "/entrypoint.sh chro…"   19 seconds ago
9af17e77670b        influxdb:latest     "/entrypoint.sh infl…"   19 seconds ago
f03d26a1a1a8        kapacitor:latest    "/entrypoint.sh kapa…"   19 seconds ago
e360ed4ccef2        telegraf:latest     "/entrypoint.sh tele…"   19 seconds ago
```

Fuente: elaboración propia, empleando *Command Shell de Microsoft Azure*

3.4.2. Configuración de *Chronograf*

Por defecto, *Chronograf* ya se encuentra instalado en la siguiente dirección Url: http://localhost:8888, se obtuvo una imagen como la siguiente:

Figura 13. **Bienvenida del programa *Chronograf***



Fuente: elaboración propia.

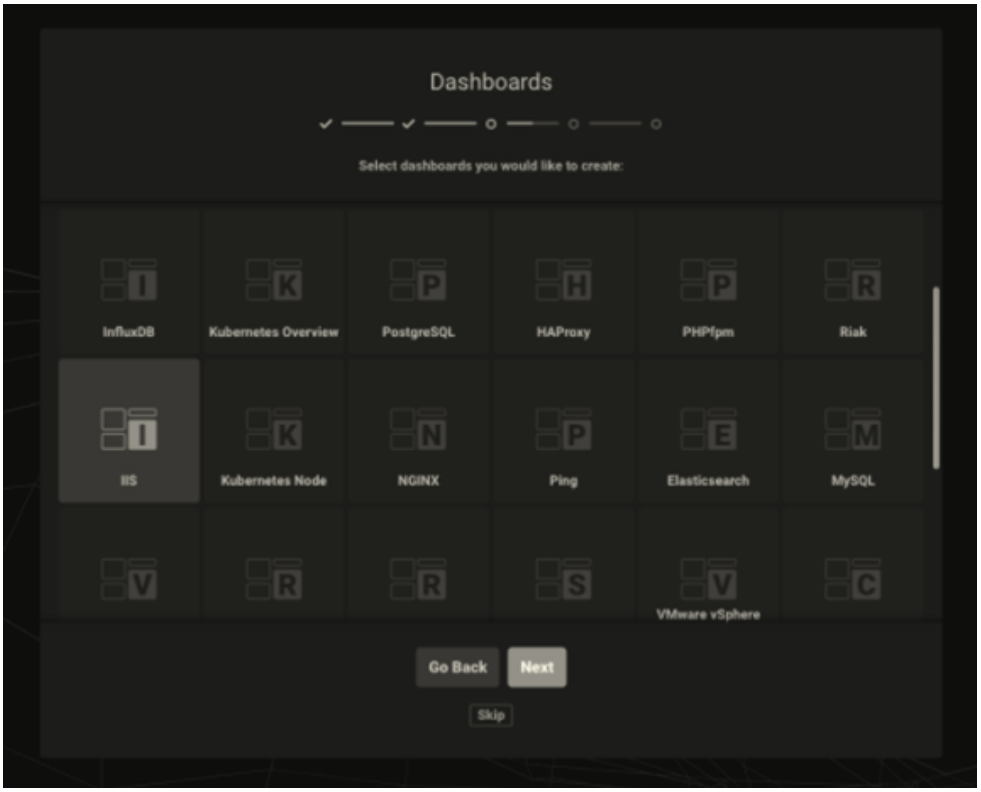
En esta pantalla se deber presionar el botón *"Get Started"*, donde luego se obtuvo la siguiente pantalla:

Figura 14. **Configuración en la conexión de *Chronograf***

Fuente: elaboración propia.

Es necesario configurar la información necesaria para conectarse a la base de datos de “*Telegraf*”, especificando el hots donde está el *InfluxDB* configurado, al finalizar de ingresar toda la información requerida, se presionó sobre el botón “Update Connection”, como siguiente paso se obtuvo la siguiente pantalla:

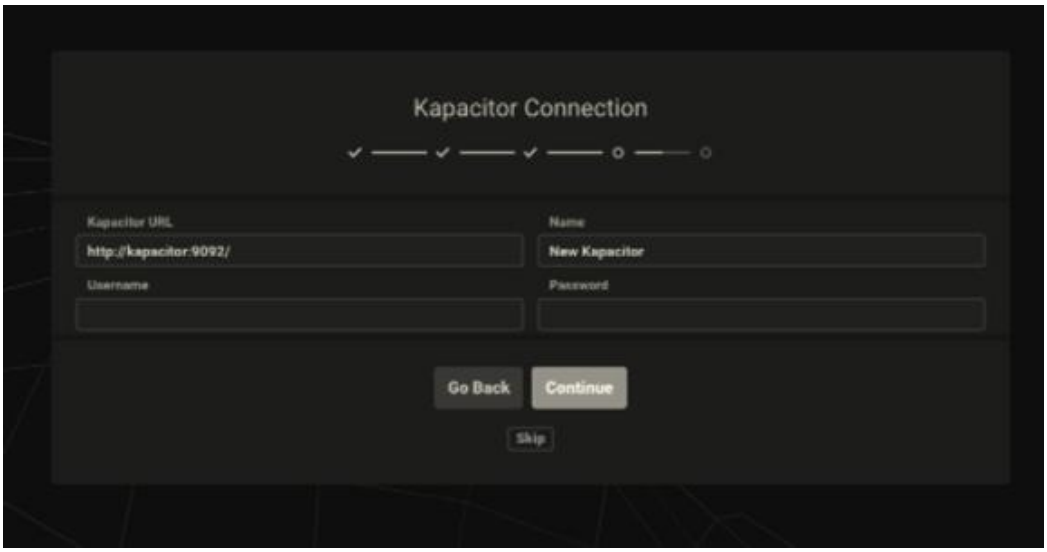
Figura 15. **Configuración de *dashboard* en *Chronograf***



Fuente: elaboración propia.

En esta pantalla se pudo configurar los diferentes estilos de gráficos tantos como fueron necesarios para tener un *dashboard* completo y llene las expectativas del usuario administrador, después el siguiente paso se configuró la conexión con el programa *Kapacitor*, se obtuvo la siguiente imagen:

Figura 16. **Configuración de conexión con *Kapacitor***



Fuente: elaboración propia.

En esta pantalla se configuraron las alertas, “*handlers*”, estas alertas fueron de mucha ayuda para llevar el control de los errores y toma de decisiones futuras.

En este punto de investigación, el sistema ya se encuentra configurado y uno de los parámetros que se ingresaron para el análisis de una de las variables fue el análisis de certificados SSL utilizados para la comunicación entre microservicios para ello fue necesario figurar *InfluxDB* y la base de datos, por la línea de comandos se ingresaron las siguientes instrucciones:

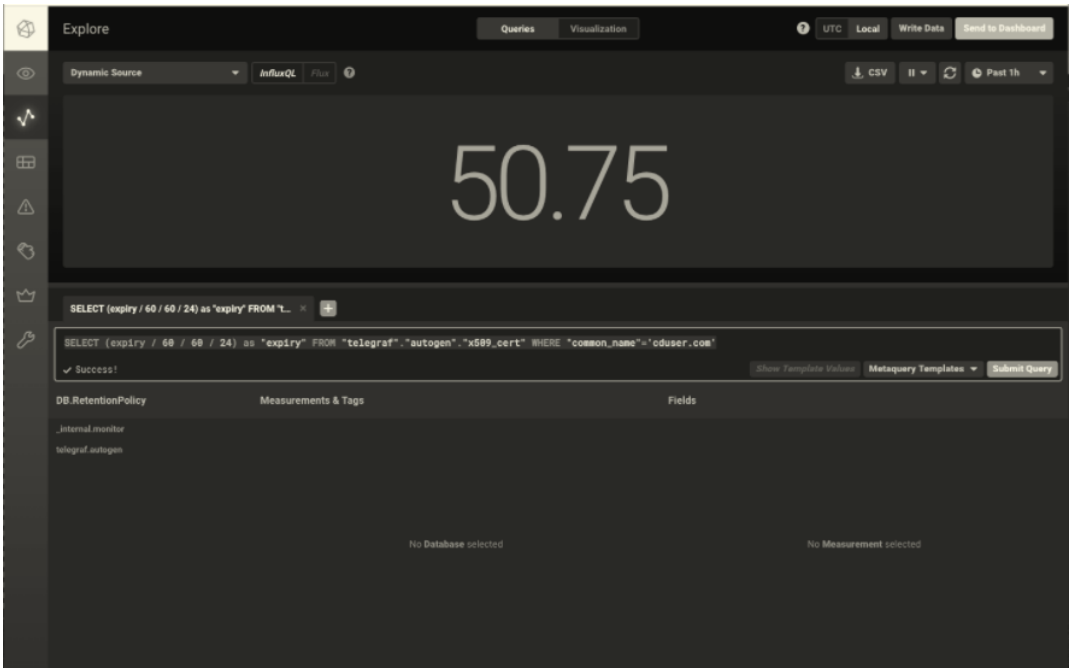
[[inputs.x509_cert]]
sources= ["https://www.example.org:443", "/etc/tls/certs/www.example.org"]

Ahora desde *Chronograf*, en el apartado de dice “explore” en la caja de texto se puede escribir instrucciones query, lo cual se escribió lo siguiente:

SELECT (expiry / 60 / 60 / 24) as "expiry" FROM "telegraf"."autogen"."x509_cert" WHERE "common_name"='cduser.com'

Interpretando el Query anterior mente escrito lo que se obtuvo fue la cantidad de días que hacen falta para que expire uno de los certificados configurados, como resultado a este query se puede observar un resultado como la siguiente imagen:

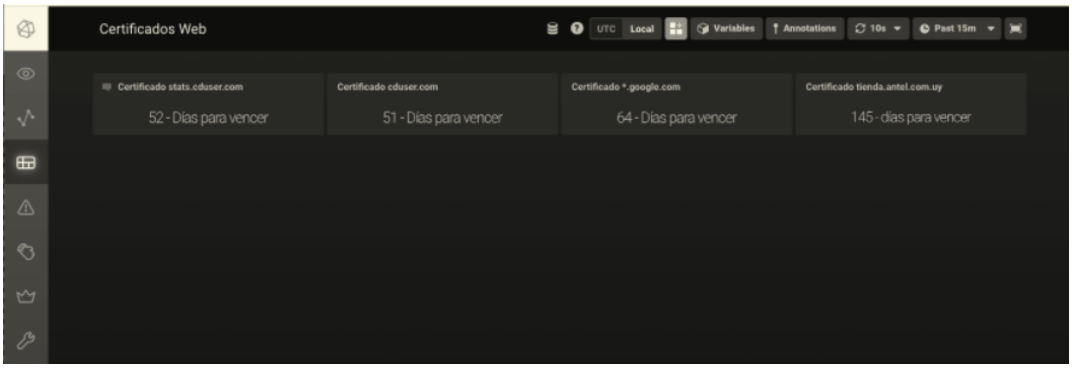
Figura 17. **Resultado de la ejecución de query**



Fuente: elaboración propia, *Ejecución Query InfluxDB.*

Dependiendo de la cantidad de certificados que se configuraron, *Telegraf* tiene la capacidad de analizar al mismo tiempo todos los certificados que se le configuren, dependiendo de la cantidad de certificados que se hayan configurado, se puede tener un *Dashboard* como el siguiente:

Figura 18. ***Dashboard* con información de cada certificado SSL**



Fuente: elaboración propia, utilizando *Dashboard Telegraf*

En el momento del cálculo del tiempo de cada certificado, el sistema observa que le quedan menos de 10 días para que se venza, entonces cambia los colores verdes por colores rojos para luego atender dicha alarma.

3.4.3. Configuración de solución *Software* en *Visual Studio* 2019

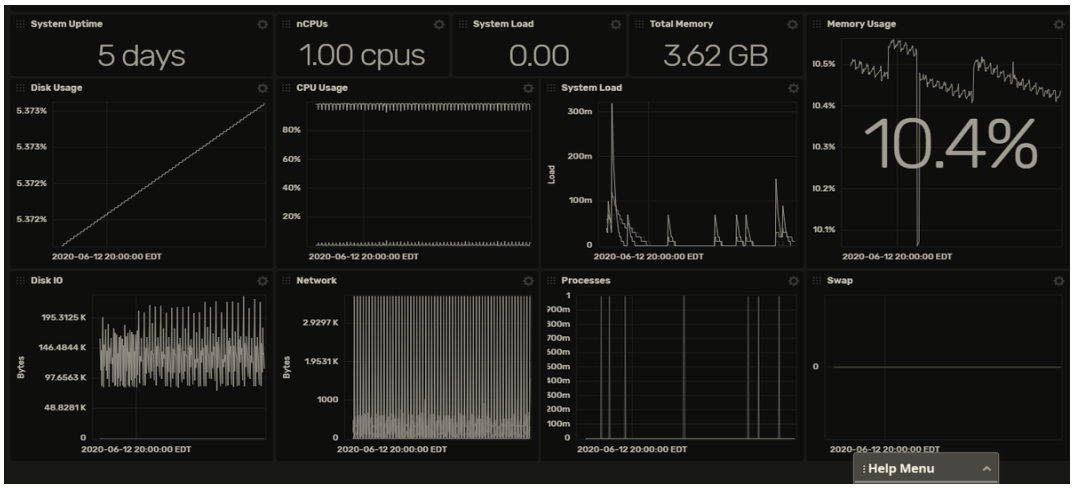
Una de las virtudes que presenta *Visual Studio* 2019, es administrar la implementación de varios microservicios almacenados en la nube de Microsoft Azure, con esta gran ventaja que ofrece esta herramienta de programación, se pudo presentar una solución que permita la automatización en la monitorización de cada uno de los elementos que componen una arquitectura de *software* basa en microservicios.

Primero se configuró una solución de *software* con características de servicio continuo, el cual se encuentra realizando consultas continuamente a la base de datos de *InfluxDB*, las consultas a la base de datos fueron para analizar el comportamiento de cada microservicio tomando como parámetros de medición las siguientes consideraciones:

- Tiempo de inactividad entre la última ejecución
- Cantidad de mensajes enviados sin error
- Cantidad de Bytes escritos a base de datos
- Cantidad de memoria utilizada en promedio
- Nivel de carga en el procesador durante su ejecución

En la siguiente imagen se muestra como quedó la implementación de los parámetros anteriormente descritos utilizando como solución grafica *Chronograf* y como base de datos *InfluxDB* y la implementación del programa en *Visual Studio* 2019 para el monitoreo automático

Figura 19. ***Dashboard* general durante el monitoreo**



Fuente: elaboración propia, utilizando *Dashboard Chronograf*

Se tomó la decisión de utilizar este tipo de *dashboard*, ya que muestra de forma gráfica, todos los parámetros de análisis que se aplicaron a cada una de las variables de la investigación, teniendo como referencia cuales eran los valores mínimos y máximos permitidos entre cada uno de los microservicios que componen el sistema, esto presenta una gran oportunidad para la toma de decisiones automáticas por medio de la solución de *software* que se implementó por medio de *Visual Studio* 2019.

Entre la toma de decisiones que el programa puede aplicar dependiendo de las circunstancias en las que el microservicio se pueda encontrar se pueden mencionar las siguientes:

- Minimizar o maximizar el uso de memoria RAM
- Minimizar o maximizar el uso de disco duro
- Reinicio del servicio completamente
- Renvío de mensajes corrigiendo el error desde el emisor

- Corrección de errores en los mensajes desde el receptor, si fuera posible

Con la implementación de este tipo de soluciones de *software* se logró alcanzar el objetivo general de esta investigación, se obtuvo un mejor rendimiento operacional, dado que el sistema se anticipa a una baja en el rendimiento de forma automática sin necesidad de la intervención humana.

4. DISCUSIÓN DE RESULTADOS

Con los resultados obtenidos se presentaron las mejores configuraciones para la implementación de una arquitectura de *software* basada en microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure. Con la tabulación de los resultados se obtuvo una estabilidad en cada uno de los microservicios, de cada 10 eventos con error, se logró anticipar y presentar una solución automática en aproximadamente 4 eventos, esto representa una mejora del 40 %.

4.1. Análisis interno

A continuación, se analizan los resultados obtenidos durante el proceso de observación sobre la solución propuesta basada en *software* para una arquitectura de microservicios para el cumplimiento de los objetivos.

Para el desarrollo de la investigación se contó con el apoyo de tutorías de parte de Microsoft, en las cuales se presentaron temas importantes e interesantes que aportaron grandemente para la elaboración de la investigación, como, por ejemplo: creación de un microservicio desde cero, configuración de mensajes entre microservicios, patrones de diseño para una arquitectura en microservicios, entre otros.

Cabe resaltar que la documentación que puede proporcionar Microsoft en su librería del conocimiento, para empezar a implementar hay que solicitar una suscripción en Microsoft Azure de estudiante, esta suscripción tiene una limitante

muy importante, únicamente se cuenta con un crédito aproximadamente USD 200, más de 25 servicios gratuitos durante 12 meses.

Es necesario ingresar una tarjeta de crédito para respaldar cualquier configuración que se haga en el portal y si se terminan los USD 200 antes de los 12 meses, se muestra un mensaje ofreciendo una suscripción de pago por consumo, la cual consiste en pagar por cada microservicio o cualquier servicio que se le asocie a los mismos, mostrando una opción de pago por consumo.

Es importante mencionar el alcance experimental del presente estudio. El registro de cada una de las soluciones aplicadas para corregir el rendimiento de un microservicio, para luego por medio de una solución de *software*, aplicar las mismas soluciones, de forma automática, tomando como criterio o variables de entrada los síntomas que el microservicio pueda estar presentando. La selección de variables depende de los criterios que determinen en la investigación y el análisis exploratorio de los datos.

Para la solución de la investigación planteada, se utilizaron soluciones *Open Source* específicas y configuradas para la monitorización de soluciones basadas en arquitecturas compuestas por microservicios.

En la primera etapa de la investigación, se aplicaron técnicas de configuración primarias para cualquier microservicio que se encuentre almacenado en la nube de Microsoft Azure, utilizando la guía en la documentación que ofrece el proveedor, ofreciendo una mejora en la generación de errores a un 0.16 % después de la 1:00 PM, según la gráfica de la figura 4.

En la segunda etapa, se aplicaron patrones de diseño distribuido, para el envío de mensajes entre cada microservicio, logrando alcanzar una estabilidad

de 300 datagramas sin error, después de observar el sistema en constante operación desde las 8 de la mañana hasta las 5 de la tarde.

Para medir la estabilidad en el envío de mensajes entre cada microservicio, se tomó como parámetro de medida, el tiempo de latencia que existe entre la solicitud de un mensaje desde el cliente hasta el primer servicio que se ejecuta en el sistema, luego el tiempo que tarda el sistema en generar la respuesta y, por último, el tiempo necesario para el envío del mismo, se realizaron varias pruebas de comunicación entre diferentes hosts físicos que ofrece Microsoft Azure, como pueden ser: Central US, West US y East US, debido a que nuestras pruebas fueron desde Guatemala, Guatemala y la distancia más corta que existe entre las tres opciones anteriores y los tiempos de respuesta fueron los más aceptados, se decidió por seleccionar Central US.

En la tercera etapa, se aplicaron nuevos patrones de diseño, pero en esta oportunidad para estabilizar la escritura y lectura de información a disco duro, al aplicar el patrón de diseño SAGA, se puede observar la mejora en la escritura y lectura a disco contra la cantidad de tiempo que le tomo escribir una cantidad de bytes, la tasa de error de escritura disminuye mejorando el porcentaje de éxito en la lectura y escritura.

En la cuarta etapa, una vez definidos los patrones de diseño necesarios, se aplicaron las configuraciones utilizando soluciones *Open Source* para el monitoreo en nuestra arquitectura, el *software* utilizado fue el siguiente: *Telegraf*, *InfluxDB*, Kapactior, *Kubernetes* y desarrollo de una solución de *software* utilizando un *framewok* de *Visual Studio 2019*, al terminar esta implementación se logró anticipar de forma automática a diferentes eventos que puede traer como resultado, dejar de funcionar completamente un microservicio y al observar los

resultados se pudo observar que por cada 10 eventos registrados, esta solución de *software* pudo resolver aproximadamente 4 eventos de forma automática.

4.2. Análisis externo

La variable principal en este trabajo de estudio es la configuración e implementación de cada microservicio que pueda formar parte en una arquitectura de *software* que cumpla estas características, debido a que uno de los problemas más complicados en este tipo de soluciones de *software* es la monitorización de cada uno de los elementos, es muy importante que el investigador cuente con un nivel de experiencia y conocimiento bastante avanzado en arquitecturas de microservicios.

Zhang en el estudio redes para la interconexión de contenedores *Dockers*, realiza y define un análisis en la comunicación y administración durante el crecimiento en la cantidad de contenedores que pueden ser administrados por *Dockers*, *Kubernetes*, *Weave*, etc. Presentando como solución al problema que existe en la comunicación entre contenedores diferentes esquemas de conexión entre ellos, planteando que esto puede ser unas de las principales causas de la falta de atomicidad en la información entre microservicios (Zhang, 2018).

Según el estudio Orquestación Automática de Contenedores, hoy en día, los marcos de orquestación de contenedores están en su infancia y no incluyen ninguna característica autonómica, *Coudify, Kubernetes*, permiten la orquestación de contenedores *Dockers*, sin embargo, como ejecutar y orquestar en un entorno distribuido sin aprovechar hipervisores sigue siendo un problema. *Docker Swarm* requiere una estática configuración de los nodos del *cluster*. Se formulan las siguientes preguntas como resultado a este documento de investigación: ¿Por qué?, ¿Cuándo?, ¿Dónde? Y ¿Cómo? Construir, cargar,

poner en marcha, migrar y apagar los contenedores bajo políticas establecidas en los comportamientos registrados (Casalicchio, 2015).

Según la investigación *Método de automatización del despliegue continuo en la nube para la implementación de microservicios*, se plantea que la arquitectura de microservicios llegó a cumplir las expectativas esperadas, sin embargo, aún se presentan desafíos como la complejidad de tener que gestionar pequeños sistemas distribuidos, la latencia de la red y la falta de fiabilidad, la tolerancia a fallas, la coherencia e integración de datos, la gestión de transacciones distribuidas, las capas de comunicación, el balanceo de carga, la orquestación, el monitoreo y la seguridad. Cada uno de los elementos que plantea Vera Rivera, hacen resaltar la necesidad que existe en reforzar cada una de las tecnologías que se utilizan para cada uno de ellos, pero como saber cuándo cada uno de ellos podría dejar de operar adecuadamente, únicamente teniendo un sistema de monitoreo bastante completo (Vera, 2016).

En la investigación *Gestión Eficiente de Arquitecturas Basadas en Microservicios* se expone que, una consecuencia del uso de microservicios como componentes, es que las aplicaciones necesitan ser diseñadas de manera que pueden tolerar fallos en el servicio. Cualquier llamada a un servicio podría fallar debido a la indisponibilidad del proveedor del servicio. El cliente tiene que responder a este evento de la mejor manera posible. Como los servicios pueden fallar en cualquier momento, es importante ser capaces de detectar fallos lo más rápido posible, si es posible automatizar la restauración del servicio. Las aplicaciones de microservicios ponen mucho énfasis en la monitorización en tiempo real de la aplicación, comprobando los elementos de la arquitectura y las métricas importantes. Pablo Roberto Cruz, concluye con la importancia que existe en la automatización para la restauración de los servicios, haciendo énfasis en la necesidad que existe actualmente ante cualquier siniestro (Cruz, 2016).

Durante el desarrollo de este trabajo de investigación, se constató que se adquirió nuevos conocimientos y experiencias, como resultado surgen nuevas ideas para innovar e implementar nuevas soluciones de *software* afrontando los retos en el crecimiento de la tecnología exige en estos últimos años y como resultado, crecimiento personal en el conocimiento.

CONCLUSIONES

1. Para mejorar la interconexión entre microservicios, es necesario comenzar con la configuración de un “Spring Boot” en Microsoft Azure, luego continuar configurando la función de descubrimiento de servicios, monitoreo y diagnóstico integrales, entre otros, con esto, se pudo observar que cada microservicio mejoró su interconexión en un promedio de 30 %, esto es aceptable en sistemas que requieran una disponibilidad de 24 horas al día, 7 días a la semana.

2. Para asegurar la comunicación entre microservicios se discutieron diferentes patrones de diseño con el fin de protegerlos ante cualquier ataque o baja en servicio, el patrón de diseño más completo y que presentó los mejores resultados fue el patrón de diseño distribuido, ya que se obtuvo estabilidad en los datagramas enviados bastante aceptable.

3. Se logró garantizar la atomicidad en los componentes de cada microservicio al implementar el patrón de diseño SAGA, fue la mejor solución que se encontró si una transacción local falla, el patrón ejecuta una serie de transacciones compensatorias que deshacen los cambios que se realizaron por las transacciones locales anteriores.

4. Para determinar una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema, se debe implementar el sistema propuesto en este documento ya que permite llevar el historial de eventos con errores, documentarlos y accionar de forma

automática justo a tiempo antes de que suceda la falla en cualquier microservicio que compone el sistema.

RECOMENDACIONES

1. En las interconexiones entre microservicios, con la configuración en Microsoft Azure, se debe considerar que tipo de membresía se va a utilizar o necesitar, ya que algunas configuraciones consumen el crédito gratuito que ofrecen y es necesario empezar a realizar una forma de pago denominada "*Pay as yo Go",* sería excelente contar con un sistema que pueda monitorizar en tiempo real el gasto de los créditos disponibles que se tienen para el uso en la nube de Microsoft Azure.

2. Para que la comunicación entre microservicios sea segura, se debe utilizar una conexión redundante, ya que, al implementar el patrón de diseño distribuido de microservicios, debido a que el servicio *Scheduler* puede fallar, y es necesario garantizar el envío de paquetes entre cada microservicio para que al momento de registrar dicha falla una nueva instancia pueda usar el punto de control y reanudar donde la instancia anterior lo dejó. Determinar las principales razones de porque el servicio Scheduler puede fallar es indispensable y con eso mejorar el envío de paquetes.

3. Se debe de considerar que la implementación del patrón de diseño SAGA no se aconseja para sistemas en los que las transacciones son muy estrechamente acopladas, debido a que estos sistemas requieren una estrecha comunicación constante y este patrón de diseño consiste en que el envío de mensajes tiene que pasar primero por cada elemento de la arquitectura hasta llegar a su destino, esto provoca lentitud en la comunicación. ¿Qué patrón de diseño puede apoyar a sistemas que si

mantienen una estrecha comunicación constante y al mismo tiempo garantizar la atomicidad en la información?

4. Al momento de implementar el sistema de monitorización en arquitecturas de microservicios, presentado en este trabajo de investigación, es necesario considerar que se utilizó un *framework* de .Net 2019 aplicando configuraciones en la nube de Microsoft Azure, su implementación se aplicó en ambientes *Windows*, es probable que en otro ambiente diferente no funcione adecuadamente, de momento no se tiene una propuesta de monitoreo automático para sistemas con arquitecturas en microservicios que garanticen la no degradación en tiempo de ejecución.

REFERENCIAS

1. Alvarado, C. (2021), *Diseño e Implementación de una Arquitectura en AWS que Garantice una Alta Disponibilidad de un Sistema Web Orientado a Microservicios para la Tabulación y Presentación de los Resultados de las Elecciones de Guatemala*. (Tesis de maestría), Universidad de San Carlos de Guatemala, Guatemala.

2. Arias Monche, E. (2016*). Demostrador IoT-Cloud en Tiempo Real.* España: Universidad Oberta de Catalunya.

3. Barrios Contreras, D. A. (2018*). Arquitectura de Microservicios, ISSN: 2344-8288 Vol 6 No. 1.* Colombia: Universidad Distrital Francisco José de Caldas.

4. Brown, K. (agosto 2016). *Más allá de Clichés: una Breve Historia de los Patrones de Microservicios*. [Mensaje en un blog]. Recuperado de https://developer.ibm.com/es/depmodels/microservices/articles/cl-evolution-microservices-patterns

5. De Paz Estrada, J. M. (2017). *Diseño e implementación de una arquitectura escalable basada en microservicios para un sistema de gestión de aprendizaje con características de red social*, (Tesis de maestría), Universidad de San Carlos de Guatemala, Guatemala.

6. Casalicchio, E. (2015). *Automatic Ochestration of Containers: Problem Definition and Research Challenges.* Suecia: Bleckinge Instiute of Technology.

7. Cilleruelo, F. (2017). *Gestor de Transacciones Distribuidas Asíncronas en Arquitecturas de Microservicios*. (Tesis de maestría). España: Universidad Nacional de Educación a Distancia Madrid.

8. Cols, F. A. (2017). *Arquitectura de Microservicios con RESTful*. (Tesis de maestría). España: Universidad Politécnica de Madrid.

9. Cruz Herrera, P. R. (2016). *Gestión Eficiente de Arquitecturas Basadas en Microservicios.* (Tesis de maestría). España: Universidad Politécnica de Valencia.

10. Echevarría, J. (2017). *Un Mundo Virtual*. España: Plaza y James 2000.

11. Fernández. L. D. (2021). *Definición de una Arquitectura Software para Diseño de Aplicaciones Web Basadas en Tecnología Java-J2EE.* España: Universidad de Oviedo Asturias.

12. Fumio, N. (2001), *Methodology to Define Software in Deterministic Manner, El Instituto de Computación Basado en metodología de Software y Tecnología.* Japón: Kadokawa Shōten.

13. Gómez Gallego, J. P. (2017). *Definición de una Arquitectura para la Transformación de Software Centralizado a Software Basado en Microservicios en el Ámbito Web*. (Tesis de maestría) Colombia: Universidad Tecnológica de Pereira.

14. Ghofrani, J. y Lübke, D. (2018). *Challenges of Microservices Architecture: A Survey on the State of the Practice*. Hannover, Alemania: Universidad de Leibniz Hannover.

15. Guanqaun, W.; Rongli, C.; Desheng, Z.; Xiaozhong, C. (2021). *Uso del Algoritmo Ant Colony en la Estrategia de Programación Basada en la Plataforma en la Nube Docker.* China: Universidad de Tecnología Dongguan.

16. Humanes, H.; Díaz, J.; Fernández, C.; Yagüe, A. (2016). *Sistemas Ciber Físicos en la Nube con Soporte a la Variabilidad y Multitenencia*. España: Universidad Politécnica de Madrid.

17. Ling-Hong, H.; Kristiyanto, D.; Bong Lee, S.; Yee Yeung, K. (2016). *Uso de Contenedores Docker con una Interfaz de Usuario Gráfica Común para Abordar la Reproducibilidad de la Investigación.* Washington, USA: Instituto de Tecnología Universidad de Washington.

18. López Hinojosa, J. D. (2017), *Arquitectura de Software Basada en Microservicios para Desarrollo de Aplicaciones Web de la Asamblea Nacional*. (Tesis de maestría). Universidad Técnica del Norte, Ibarra Ecuador.

19. Mouat, A. (2016). *Using Docker Developing and Deploying Software with Containers.* USA O'Reilly, NY, USA: Allitebooks

20. Oliveria, A.; Suneja Sahil, N.; Nagpurkar, P.; Isci, C. (2017). *A cloud-Native Monitoring an Analytics Framework.* NY, USA: IBM Research Division.

21. Ruelas Acero, D. A. (2017). *Modelo de Composición de Microservicios para implementación de una Aplicación Web de Comercio Electrónico utilizando Kubernetes.* Perú: Universidad Nacional del Altiplano.

22. Saman, B. (2017). *Monitoring and Analysis of Microservices Performance*. Kurdistan: University of Duhok.

23. Vera Rivera, F. (2017). *Método de automatización del despliegue continuo en la nube para la implementación de microservicios*. Cali, Colombia: Universidad Francisco de Paula Santander.

24. Zhang, M. (2018), *Estudio y Análisis de proyectos sobre redes para la interconexión de contenedores Docker*. (Tesis de maestría). Universidad Politécnica de Valencia. España.

APÉNDICES

Apéndice 1. **Matriz de coherencia**

Tema: “Una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la ciudad de Guatemala, Guatemala”

Problema	Objetivos	Metodología	Conclusiones	Recomendaciones
Problemas Especificos	**Objetivos Específicos**	"Enfoque de la investigación: El tipo de estudio que se utilizó fue estadístico matemático, por sus características, es el que mejor aportó información a la investigación, Tipo de la investigación: El tipo de investigación que se aplicó fue de tipo descriptivo, debido a que fue necesario analizar de forma individual cada contenedor, determinar causas y razones por las cuales uno de ellos puede dejar de funcionar totalmente. Diseño de la investigación: Con la información recolectada y analizada, se realizaron pruebas para luego volver a analizar el resultado y los posibles nuevos comportamientos de cualquier contenedor, con este ejercicio constante, se buscó predecir futuras bajas en el rendimiento de uno o varios microservicios y establecer un mejor uso y rendimiento de los recursos de cada uno de ellos, para esto se aplicó un tipo de diseño experimental. Variables: Las variables son Microservicios y Contenedores, con el fin de obtener resultados para el análisis de la investigación. Unidad de Análisis: La población en estudio fue un conjunto de microservicios que componen un sistema configurado en la Nube de Microsoft Azure, cada microservicio fue analizado de forma individual durante los meses de julio y agosto de 2021. Fase 1. Revisión literaria Fase 2. Recolección de la información Fase 3. Análisis de la información Fase 4. Interpretación de la información Fase 5. Redacción del informe final"		
Inexistencia de la configuración e interconexión de una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de micro servicios con ayuda de soluciones open source para monitoreo en la Ciudad de Guatemala, Guatemala.	Establecer la configuración e interconexión de una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de micro servicios con ayuda de soluciones open source para monitoreo en la Ciudad de Guatemala, Guatemala.		Para asegurar la comunicación entre microservicios se discutieron diferentes patrones de diseño con el fin de protegerlos ante cualquier ataque o baja en servicio, el patrón de diseño más completo y que presento los mejores resultados fue el patrón de diseño distribuido, se obtuvo una estabilidad en los datagramas enviados bastante aceptable	En las interconexiones entre microservicios, con la configuración en Microsoft Azure, se debe considerar que tipo de membresía se va a utilizar o necesitar, ya que algunas configuraciones consumen el crédito gratuito que ofrecen y es necesario empezar a realizar una forma de pago denominada “*Pay as yo Go”*
Inexistencia de la configuración e interconexión de una solución de software que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones open source para monitoreo en la Ciudad de Guatemala, Guatemala.	Determinar cómo se asegura la comunicación entre microservicios ante cualquier ataque de una solución de *software* que garantiza la no degradación en el tiempo de ejecución de los elementos del sistema en una arquitectura de microservicios con ayuda de soluciones *open source* para monitoreo en la nube de Microsoft Azure, en la Ciudad de Guatemala, Guatemala		Se logro garantizar la atomicidad en los componentes de cada microservicio al implementar el patrón de diseño SAGA, fue la mejor solución que se encontró, si una transacción local falla, el patrón ejecuta una serie de transacciones compensatorias que deshacen los cambios que se realizaron por las transacciones locales anteriores	Para que la comunicación entre microservicios sea segura, se recomienda utilizar una conexión redundante, ya que, al implementar el patrón de diseño distribuido de microservicios, debido a que el servicio *Scheduler* puede fallar, y es necesario garantizar el envío de paquetes entre cada microservicio para que al momento de registrar dicha falla una nueva instancia pueda usar el punto de control y reanudar donde la instancia anterior lo dejó

Fuente: elaboración propia.