Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

IMPLEMENTACIÓN DE *CHAOS ENGINEERING* EN UN ENTORNO CLOUD NATIVE

Jossie Bismarck Castrillo Fajardo
Asesorado por el Ing. Sergio Arnaldo Méndez Aguilar

Guatemala, octubre de 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

IMPLEMENTACION DE *CHAOS ENGINEERING* EN UN ENTORNO CLOUD NATIVE

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

JOSSIE BISMARCK CASTRILLO FAJARDO
ASESORADO POR EL ING. SERGIO ARNALDO MÉNDEZ AGUILAR

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, OCTUBRE 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 Armando 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 Cordoba Estrada
EXAMINADOR	Ing. Marlon Francisco Orellana López
EXAMINADOR	Ing. Neftali De Jesús Calderón Méndez
EXAMINADOR	Ing. Gabriel Alejandro Díaz López
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:

IMPLEMENTACION DE *CHAOS ENGINEERING* EN UN ENTORNO CLOUD NATIVE

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería En Ciencias y Sistemas, con fecha 20 de julio de 2021.

Jossie Bismarck Castrillo Fajardo

Guatemala, 07 de agosto de 2022

Ingeniero
Carlos Alfredo Azurdia
Coordinador de Privados y Trabajos de Tesis
Escuela de Ingeniería en Ciencias y Sistemas
Facultad de Ingeniería - USAC

Respetable Ingeniero Azurdia:

Por este medio hago de su conocimiento que en mi rol de asesor del trabajo de investigación realizado por el estudiante **JOSSIE BISMARCK CASTRILLO FAJARDO** con carné **201313692 y CUI 2338 00514 0101** titulado **"IMPLEMENTACIÓN DE CHAOS ENGINEERING EN UN ENTORNO CLOUD NATIVE"**, luego de corroborar que el mismo se encuentra finalizado, lo he revisado y doy fé de que el mismo cumple con los objetivos propuestos en el respectivo protocolo, por consiguiente, procedo a la aprobación correspondiente.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

DocuSigned by:
E8B86BE7CBBA447...

Ing. Sergio Arnaldo Méndez Aguilar
Colegiado No. 10958

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala 11 de agosto de 2022

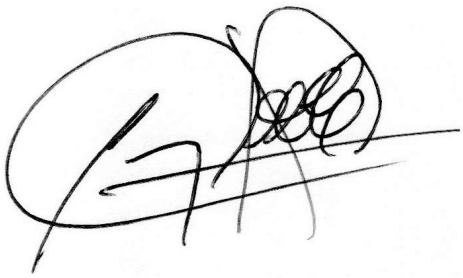
Ingeniero
Carlos Gustavo Alonzo
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Alonzo:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **JOSSIE BISMARCK CASTRILLO FAJARDO** con carné **201313692** y CUI **2338 00514 0101** titulado **"IMPLEMENTACIÓN DE CHAOS ENGINEERING EN UN ENTORNO CLOUD NATIVE"** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo aprobado.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,



Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

LNG.DIRECTOR.197.EICCSS.2022

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del Asesor, el visto bueno del Coordinador de área y la aprobación del área de lingüística del trabajo de graduación titulado: **IMPLEMENTACIÓN DE CHAOS ENGINEERING EN UN ENTORNO CLOUD NATIVE**, presentado por: **Jossie Bismarck Castrillo Fajardo**, procedo con el Aval del mismo, ya que cumple con los requisitos normados por la Facultad de Ingeniería.

"ID Y ENSEÑAD A TODOS"

Msc. Ing. Carlos Gustavo Alonzo
Director
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, septiembre de 2022

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

LNG.DECANATO.OI.665.2022

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 Ingeniería en Ciencias y Sistemas, al Trabajo de Graduación titulado: **IMPLEMENTACIÓN DE CHAOS ENGINEERING EN UN ENTORNO CLOUD NATIVE,** presentado por: **Jossie Bismarck Castrillo Fajardo,** 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, octubre de 2022

AACE/gaoc

ACTO QUE DEDICO A:

Dios

Por haberme dado todos estos años la fortaleza y la perseverancia necesaria para no rendirme, incluso cuando dejé de creer en mí y pensé que ya no podía continuar.

Mis abuelitos

Loida Ventura, por haber sido como una madre a lo largo de mi vida, por haber ayudado a criarme y haberme dado las bases morales para ser la persona que hoy soy y Manuel Fajardo (q. e. p. d.) por haber sido como un segundo padre y por haberme enseñado como ser un caballero, responsable por mi familia y mi hogar, sé que le hubiera encantado verme alcanzar este logro.

Mis padres

Wendy Fajardo por enseñarme que todo es posible con perseverancia y que sin importar cuantas veces caiga siempre me debo levantar nuevamente y Arnoldo Castrillo por ser una fuente de inspiración para seguir siempre adelante, hace muchos años cuando se graduó me dijo que algún día yo también estaría saliendo de la facultad y recorrería esos pasillos, tenía razón.

Mi hermana Wendy Castrillo por apoyarme siempre y animarme con sus bromas en los momentos de estrés, espero haber sido un buen ejemplo para seguir.

Mi tía Mirna Fajardo, por haber estado siempre ahí para mí, llevándome cuando ella aún estaba en la universidad a sus clases y haber despertado en mí ese deseo de estudiar para alcanzar mis metas desde que era niño. Por haberme apoyado siempre en mis decisiones académicas, laborales y personales, por haber sido una guía durante todo mi recorrido como estudiante y una tercera madre en mi vida.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por ser la casa de estudios que me formo como un profesional ético, responsable y de calidad.
Facultad de Ingeniería	Por haberme brindado momentos muy buenos durante mí formación y permitirme conocer personas maravillosas.
Mi familia	Loida Fajardo; Manuel Fajardo, Wendy Fajardo, Arnoldo Castrillo, Wendy Castrillo y Mirna Fajardo, a todos por darme siempre su apoyo y motivación para no rendirme nunca.
Ing. Sergio Méndez	Por ser un catedrático excepcional y presionarme para lograr hacer algo de la talla correspondiente. No cualquier catedrático insta a sus alumnos a presionarse para lograr algo superior, incluso brindarme la oportunidad de poder mostrar mis avances a nivel internacional.
Omar Pérez	Por ser un gran amigo desde el inicio de nuestra carrera; molestando cada día, brindando apoyo y consuelo en los momentos difíciles, incluso ahora que nuestra amistad trasciende las aulas.

Saraí De León	Por ser una maravillosa amiga a lo largo de los años; bromeando, dando palabras de aliento, motivación para seguir adelante y su apoyo incondicional en todo momento.
Ariel De León	Por ser siendo un gran amigo desde nuestra niñez y acompañarme por este camino, aun ahora luego de un par de décadas continuamos con la amistad.
Athena Solís	Por convertirse en una amiga maravillosa y considerada, siempre con una actitud positiva.
Mis amigos	Byron Chicara, Carlos Orantes, Luis Rivera, David Morales, Glen Calel, Diego Momotic, Luis Salazar, Katherine Serrano, Carlos Peralta, Roberto Cux por haberse convertido en personas tan importantes y grandiosas en mi vida, no solo dentro de las aulas.
Mis jefes	Ingrid Valladares, Pablo Girón, Elieth Jacobo y Heber Pernilla por haberme dado la oportunidad de desarrollarme y terminar mis estudios cuando necesite el tiempo para concluir el proceso universitario, así como orientarme en el ámbito profesional.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
ms	Milisegundos.
N/A	No aplica.
%	Porcentaje.
s	Segundos.
~$	Utilizado para indicar que el texto a continuación es una porción de código para agregar desde la consola de un sistema; simula el inicio de línea en una terminal.

GLOSARIO

API	Es una interfaz de programación de aplicaciones por sus siglas en inglés, este conjunto de rutinas provee acceso a funciones de un determinado software.
Aplicación	Es un programa informático diseñado para realizar alguna función específica.
AttrOverride	Es un experimento que busca sobre escribir los atributos de ciertos procesos y programas en ejecución.
Bare metal	Es el *hardware* puro.
Blackhole	Es un experimento que bota todo el tráfico entrante en una red.
Cache	Es un tipo de memoria que almacena información dinámicamente, esta información normalmente es transitoria, no permanente.
Clúster	Es un conjunto de nodos que ejecutan aplicaciones contenerizadas.
CNCF	Cloud Native Computing Foundation.

Concurrencia	Es la propiedad de los sistemas para ocurrir al mismo tiempo.
Container kill	Es un experimento que busca “matar” un contenedor en ejecución.
Data	Una variable cuantitativa o cualitativa para describir algún hecho.
DNS	Es el sistema de nombres de dominio, el cual funciona como un diccionario que relaciona direcciones IP con nombres de dominio.
Docker *service kill*	Es un experimento que busca detener la ejecución de Docker y todo lo que tenga en ejecución.
Downtime	Se refiere a un periodo de tiempo en el cual un sistema o servicio no está disponible.
Fault Injection	Es una técnica para hacer que un sistema se comporte de formas inesperadas, generando fallas internas.
Framework	Es un entorno de trabajo el cual esta estandarizado de alguna forma por ciertos conceptos, prácticas y criterios para afrontar una problemática.
GCP	Es la plataforma de trabajo de Google Compute Cloud

Health-check	Es una petición enviada para identificar problemas comunes y diagnosticarlos.
Holístico	Indica que los sistemas y sus componentes se deben analizar como un todo y no por las partes que lo componen individualmente.
HTTP	Protocolo de transferencia de hipertexto**.**
Incubating	Es un nivel de clasificación de proyectos en CNCF considero intermedio, en el cual se valida la madurez del proyecto.
Infraestructura	Es el conjunto de servicios e instalaciones necesarias para realizar una actividad.
Instancia	Es la computadora o servidor individual que se utiliza.
IO	Es un nombre utilizado para entrada/salida de datos en un sistema.
Jitter	Es la fluctuación del retado durante el envió de una señal digital, o ruido en una señal digital.
KPI	Key performance indicador, son las métricas utilizadas para diferentes mediciones de interés.
Latencia	Es la suma de los retardos dentro de una red, se produce por la demora en él envió de paquetes.

Métrica	Es la medida con la cual se establece que se calculara alguna magnitud.
Middleware	Es un software intermedio entre un sistema operativo y aplicación que permite la comunicación de aplicaciones distribuidas.
MTBF	*Mean time between failures* o tempo medio entre fallas.
MTTF	*Mean time to fail* o tiempo medio para fallar.
MTTR	*Mean time to restore* o tiempo medio de recuperación.
Network bandwidth action	Es un experimento que limita el ancho de banda de una red.
Network corrupt	Es un experimento que corrompe los paquetes que se envían en la red.
Network delay	retraso en él envió de paquetes en una red.
Network duplicate	Duplicación de paquetes en una red.
Network loss	Perdida de paquetes en una red.
Network partition Action	Supera el comportamiento tolerable en una partición de red.

Nodo	Es un punto de conexión para vacíos elementos o instancias.
On premise	Se refiere a poseer el hardware físicamente en una ubicación y no en la nube.
OpenShift	Es una plataforma para el manejo de contenedores.
Open source	Se refiere al código abierto, es un código diseñado para que sea accesible al público, cualquier persona puede ver, modificar y distribuir el código libremente.
Orquestación	Es la automatización y escalabilidad en la implementación y administración de contenedores.
Out-of-the-box	Se refiere al balanceo de carga dinámico entre instancias activas.
PaaS	Platform as a service o Plataforma como servicio.
Pipeline	Es una técnica que se utiliza para implementar tareas simultaneas o consecutivas en un orden determinado.
Plugin	son aplicaciones que agregan funcionalidad al software existente***.***
POD	Representa una instancia en un clúster de Kubernetes, es la unidad más pequeña que se puede implementar en estos.

Pod* CPU *hog	Es un experimento que busca llevar al límite la utilización de CPU de un *pod*, haciéndolo lento.
Pod delete	Es un experimento que busca la eliminación de un *pod*.
Pod failure-metrics	Es un experimento que busca llevar todas las métricas de un *pod* a un estado fallido.
Pod kill	Es un experimento que busca “matar” un *pod*.
Pod network latency	Es un experimento que busca generar latencia en la red interna de los *pod*.
Pod network loss	Es un experimento que busca generar perdida de paquetes dentro de la red de un *pod* o su comunicación con otros.
Pool ejection	Es una estrategia usada con resiliencia, en la cual, si alguna petición a alguna instancia falla, se excluye temporalmente de los accesos.
Proceso	Es un grupo de pasos consecutivos para un fin especifico.
Proxy	Es una red intermediaria entre las peticiones a ciertos recursos que realiza un cliente a un servidor.

Red	Es un conjunto de hardware y software conectados entre sí a través de medios físicos o inalámbricos.
Reliability	Fiabilidad
Resiliencia	Es la capacidad de adaptarse a situaciones adversas.
Rollback	En el ámbito de manejo de arquitecturas y software se refiere a un proceso en el cual se descartan y revierten las actualizaciones realizadas en un determinado punto en el tiempo hasta regresar a una versión considerada estable.
Restful	Es una forma de diseño arquitectónico de software utilizado en sistemas alojados en internet en el cual se manejan peticiones HTTP.
Retry	Es el reenvío de una petición luego de que la original haya presentado errores.
RPC	*Remote procedure call* o llamada a procedimiento remoto.
Sandbox	Es un nivel de clasificación de proyectos en CNCF considero intermedio, en el cual se valida la madurez del proyecto.
Script	Es una secuencia de órdenes para ejecutarse.

SDK	*Software development kit* por sus siglas en inglés, es un grupo de herramientas que permite la creación de una aplicación para un sistema concreto.
Service discovery	Descubrimiento de servicios.
Simple circuit breaker	Realiza verificaciones sobre los *pods* con Kubernetes, si después de X errores consecutivos no hay uno correcto el *pod* es removido del grupo de balanceo de carga.
Sistema	Es un conjunto de elementos que funcionan en grupo y se relacionan con al menos uno de los otros componentes.
Sistema operativo	Es el software principal que se encarga de gestionar los recursos de hardware.
SOA	*Service oriented architecture* o arquitectura orientada a servicios.
TCP	Este es un protocolo para el control de transmisión de datos, garantiza que los datos serán entregados a su destino sin error y en el orden en el cual se enviaron.
Timeout	es un evento que ocurre luego de un tiempo determinado.

TLS	*Transport layer security*, es una versión actualizada del certificado SSL, el cual es un estándar para una conexión segura en la red.
Traffic governance	Gobernanza del tráfico, referente a la red.
Traza	Es una captura del tráfico que pasar por una red en un periodo de tiempo.
Trigger	Es un disparador de eventos, es decir cuando estos se activan, se ejecuta alguna acción establecida.
Vulnerabilidad	El riesgo a fallos de un sistema.

RESUMEN

Se busca definir una guía para implementar *chaos engineering* en sistemas *cloud native*, que pueda utilizarse o tomarse como referencia para la aplicación de este tipo de pruebas realizadas sobre las infraestructuras que cumplan con las características de *cloud native*, se podrá tomar correctamente la decisión de qué tipo de inyección de caos realizar en diferentes componentes de la infraestructura. En base a la realización de pruebas constantes los usuarios podrán identificar potenciales fallas en sus sistemas en base a un estado saludable para poder tomar la decisión de mejorar su infraestructura o considerarlo como una falla aceptable para sus fines.

OBJETIVOS

General

Definir una guía para la implementación de *chaos engineering* en sistemas *cloud native* definiendo sus bases y explorando herramientas para realizar pruebas y la ejecución de experimentos en base a un ciclo continuo de ejecución de caos.

Específicos

1. Definir las bases y teoría que dieron origen a *chaos engineering*.

2. Definir los principios asociados a *chaos engineering*.

3. Identificar herramientas existentes para trabajar con *chaos engineering*.

4. Comparar las ventajas y desventajas de la aplicación de *chaos engineering* para el contraste de la importancia de su aplicación.

5. Identificar costos estimados asociados a la implementación de *chaos engineering*.

INTRODUCCIÓN

La presente investigación se refiere al tema de la aplicación de *chaos engineering* en un entorno *cloud native*, el cual se puede definir como la aplicación de diferentes experimentos y pruebas sobre cualquier componente de una arquitectura informática nativa de la nube, para generar de forma intencional fallas que aún no se han descubierto, esto con la finalidad de identificar vulnerabilidades potenciales para su corrección y prevención de caídas no contempladas en los diferentes sistemas.

Para el análisis de esta disciplina es necesario comprender sus antecedentes históricos, los cuales dieron origen a una disciplina relativamente nueva. Los diseños arquitectónicos y su evolución a través del uso de servicios, microservicios hasta un nivel más complejo con *service mesh* cuyo uso generó una nueva problemática, la de mantener un sistema confiable y seguro en continuo funcionamiento.

La investigación de esta disciplina se realizó por el interés de buscar una forma definida y detallada de realizar experimentos de *chaos engineering* en entornos *cloud native*, siendo estos los que están tomando el auge para brindar servicios en una cultura donde la disponibilidad de los sistemas es una necesidad, de esta forma estableció una serie de pasos que pueden seguirse para realizar experimentos controlados y detectar vulnerabilidades potenciales, pero aún desconocidas por la aleatoriedad en la naturaleza de los sistemas.

En el capítulo uno se realizó la definición de lo que *chaos engineering* es y las bases que dieron origen a la necesidad de su aplicación en los sistemas modernos. Esto combinado con los principios que conforman esta disciplina, la explicación de ¿En dónde se puede aplicar caos dentro de los sistemas?, así también las ventajas y desventajas que apoyan a la decisión de su uso.

En el capítulo dos se definen los sistemas distribuidos orientados a servicios y microservicios en conjunto con sus características y la problemática que da origen al *service mesh*, el modelo arquitectónico utilizado para la generación de los entornos *cloud native*, en donde entran al juego nuevos patrones de diseño, herramientas y características que se deben cumplir para adaptarse a este modelo de plataformas.

En el capítulo tres se realizó la definición teórica *cloud native* identificando su concepto y la organización que se encarga de generar los estándares y graduación de proyectos a nivel mundial que colaboran con el desarrollo de este modelo. De la misma forma se detallan las practicas utilizadas para desplegar un sistema y que este sea considerado *cloud native*.

En el capítulo cuatro se describen algunas de las herramientas utilizadas para poder desarrollar experimentos de *chaos engineering*, cada una con sus funciones u objetivos principales.

En el capítulo cinco se define una guía con lo necesario para poder realizar experimentos de *chaos engineering*, definiendo las condiciones base necesarias para tener un entorno estable sobre el cual experimentar, se definió una arquitectura, estado saludable y un experimento a realizar para ejemplificar el paso a paso para realizar un experimento inicial con Chaos Mesh y Linkerd mediante el ciclo de aplicación de *chaos engineering*. Posteriormente se detalla un experimento y el análisis de sus resultados para la toma de decisiones en un ambiente real.

En el capítulo seis se muestra la popularidad actual de esta disciplina mediante la recopilación de datos estadísticos a través de una encuesta realizada a personas que se desenvuelven en el área informática, contrastando con el largo camino que aún tiene por delante una disciplina tan joven, con beneficios que aún falta por explotar.

1. CHAOS ENGINEERING

Es una disciplina que nos ayuda a realizar pruebas y experimentos sobre diferentes sistemas o arquitecturas informáticas, sus bases se originan con la teoría del caos, considerando que ningún sistema puede permanecer inmutable o ser completamente inmune a fallas, lo mismo ocurre con los sistemas informáticos, entre más extensos son más complicado se vuelve predecir su comportamiento para lo cual *chaos engineering* entra en el juego.

1.1. Antifragilidad

La antifragilidad es un concepto poco manejado y subestimado en algunas ocasiones, se puede ver como una habilidad que todas las cosas tienen para mejorar u obtener algún beneficio en base al desorden, la irregularidad de las cosas, el azar y el caos natural que se genera en cualquier situación. Cuando nos enfermamos obtenemos mayores defensas ante una enfermedad y resultamos beneficiados al obtener una mejora por la enfermedad, los sistemas y entornos artificiales creados por el hombre como lo son los sistemas informáticos están propensos al mismo principio, al ser afectados por cualquier situación negativa, podemos mejorarlos y hacerlos más resistentes tomando esto como una ventaja para el progreso de dichos sistemas.

1.2. Caos

La teoría del caos surge por el meteorólogo y matemático Edward Lorenz a mediados del siglo XX, que estudia algunos sistemas dinámicos que evolucionan con el tiempo, muy sensibles a variar de sus condiciones iniciales,

donde cada pequeño cambio en estas condiciones genera comportamientos completamente distintos, haciendo que la predicción de ciertas condiciones a futuro sea una tarea difícil de manejar a largo plazo.

Otra forma con la que se le conoce es como el Efecto Mariposa y su conocido dicho de "*el aleteo de las alas de una mariposa puede provocar un tsunami al otro lado del mundo*", utilizado como una forma drástica de decir que cualquier variación en las condiciones de un sistema traerá un cambio totalmente distinto en el proceso y su resultado final.

1.3. Concepto

En base a los conceptos anteriores nace la idea que da origen a *chaos engineering*, aunque esta aún es considerada una disciplina relativamente nueva en el ámbito informático. Según *principles of chaos.org*, "La ingeniería del caos es la disciplina de experimentar en un sistema, con la finalidad generar confianza en la capacidad del sistema para soportar condiciones turbulentas en producción".[1].

Esta es la idea central que pone en Jaque la importancia de *chaos engineering*, jugar con nuestros sistemas para poder medir sus capacidades y limitantes ante condiciones adversas, muchos profesionales en el área informática aún son escépticos sobre llevar al borde de un precipicio a sus sistemas si estos ya funcionan correctamente, claro, están en lo correcto los sistemas funcionan correctamente en condiciones ideales, no obstante, no están exentos de fallas.

[1] Community, Chaos. *Principios de la Ingeniería del Caos*. https://principlesofchaos.org/. Consulta: 5 de marzo de 2021.

El punto central de jugar con nuestros sistemas para inducir fallos a propósito es eso, identificar esas fallas aún escondidas en un comportamiento normal, para poder fortalecer nuestro sistema y que esté preparado para situaciones, que, aunque sean poco comunes, media vez exista una pequeña probabilidad de ocurrencia, se debe tomar medidas preventivas asumiendo que pasara.

1.4. Historia de *chaos engineering*

La historia de *chaos engineering* empieza a formarse como concepto a inicios de la década pasada en el año 2011, Netflix se encontraba trabajando en la migración de sus sistemas a la nube, tomando en cuenta que en la nube se debe tener un objetivo orientado a la redundancia y tolerancia a fallos, Yuri Izraeilevsky[2], Director de Infraestructura de Nube y Sistemas, narra que debían tener una arquitectura en la que se considerara que cada componente individual podía fallar sin afectar la disponibilidad del resto del sistema.

Buscaban tener una arquitectura más fuerte que su eslabón más débil, pero consideraron que diseñar una arquitectura con tolerancia a fallos no era suficiente, querían desafiar constantemente su capacidad de perseverar ante fracasos. Por tal motivo Greg Orzell decidió abordar la falta de pruebas de resiliencia manejando una herramienta que podría causar fallos en el ambiente en producción que ya tenían, es decir utilizando el entorno que manejaban los clientes del gigante del streaming. Buscaron utilizar un modelo en el que se consideraba que no había fallas como uno en el que se debía pensar que las fallas serían inevitables, esto para tomar la resiliencia como una obligación para ellos y no una opción.

[2] CAREY, Scott. *¿Qué es Chaos Monkey? La ingeniería del caos, explicada*. https://cioperu.pe/articulo/30319/que-es-chaos-monkey-la-ingenieria-del-caos-explicada/?p=3. Consulta: 5 de marzo de 2021.

Sus pruebas se basaron en matar instancias activas de forma aleatoria en periodos regulares de tiempo, probando su arquitectura redundante generando fallas y comprobando que estas no afectarán de manera visible a sus clientes. Para esto crearon la herramienta Chaos Monkey, una de las primeras herramientas para generar caos, que estaba orientada a deshabilitar a propósito las instancias en su red para verificar cómo responden los sistemas restantes a este fallo, esta herramienta ahora es parte de Simian Army, uno de los grupos de herramientas más grandes que hay, diseñado para simular respuestas a diversos fallos en los sistemas, muchos de estos fallos extremos.

En 2015, Bruce Wong[3] decidió crear un equipo orientado a la ingeniería de caos dentro de Netflix, con la misión de crear un camino dentro de esta disciplina, generando el concepto oficial de *chaos engineering*, ellos buscaban que sonara bien y Casey Rosenthal a quien se le encargó esta labor, desarrollaron los principios de *chaos engineering* para formalizarlo oficialmente dentro de 5 principios generales para poner en práctica.

A partir de ese punto han surgido diferentes herramientas y organizaciones que han brindado sus aportes para tener mayores avances en esta disciplina debido a los beneficios que brinda si se sabe manejar, actualmente CNCF, la fundación que impulsa los sistemas *cloud native* y sus diferentes embajadores se encargan de buscar un mayor auge y concientización para el uso de esta disciplina en entornos de la nube para brindar la menor cantidad de visibilidad de fallas para los clientes finales de los sistemas.

[3] CAREY, Scott. *¿Qué es Chaos Monkey? La ingeniería del caos, explicada*. https://cioperu.pe/articulo/30319/que-es-chaos-monkey-la-ingenieria-del-caos-explicada/?p=3. Consulta: 5 de marzo de 2021.

1.5. Principios del caos

Estos llamados principios de *chaos engineering* muestran una forma de aplicación ante un entorno ideal que se pueden aplicar en el proceso de experimentación, estos principios son una base de cómo se puede aplicar esta disciplina en un sistema distribuido a cualquier escala.

1.5.1. Construir una hipótesis sobre el comportamiento de un estado normal

La forma en la que se maneja normalmente cualquier experimento es definiendo de primero una hipótesis. Los principios buscan construir una hipótesis en base a la definición de un estado saludable o normal del sistema. Esto significa que debemos enfocarnos en la forma en que esperamos que el sistema se comporte y monitorear ese comportamiento. De esta forma los ingenieros deben dejar por un lado el código y enfocarse en la salida holística del sistema. Esto lleva el proceso a través de la verificación y validación.[4]

Muchas veces se intenta encontrar la raíz del problema y tratar de entenderlo por reducción de causas, haciendo que se tenga que explorar más a fondo, distrayendo de la mejor forma de aprendizaje que *chaos engineering* puede brindarnos, es decir enfocarse en KPIS u otras métricas que dejan claro las prioridades del negocio haciéndolas los mejores estados saludables de un sistema.

Al centrarse en la salida final de un sistema en vez de sus partes internas, se pueden tomar estas salidas durante un determinado periodo de tiempo en un

[4] Community, Chaos. *Principios de la Ingeniería del Caos*. https://principlesofchaos.org/. Consulta: 5 de marzo de 2021.

estado saludable, usualmente se manejan periodos cortos de tiempo. El rendimiento de nuestro sistema, tasas de fallos, latencia, tiempos de respuesta, entre otros. Son las mejores métricas guiadas por los intereses del negocio, al centrarse en la forma en la que un sistema se comporta durante los experimentos, el caos comprobará que el sistema funciona correctamente en lugar de intentar validar cómo funciona el sistema en sí.

1.5.2. Variar los eventos del mundo real

Este principio establece que las variables en los experimentos deben reflejar eventos del mundo real. "Se deben priorizar los eventos ya sea por impacto potencial o frecuencia estimada". Se debe considerar a cualquier evento que pueda interrumpir el estado saludable de un sistema como una variable potencial en un experimento de caos.[5]

1.5.3. Ejecutar experimentos en producción

Por medio de la experimentación se puede aprender sobre el sistema que se está monitoreando. Al realizar pruebas en un entorno local o interno, se genera seguridad y comprueba el buen funcionamiento del sistema en ese entorno, no obstante, las condiciones de un entorno de pruebas contra un entorno en producción jamás serán igual, hay demasiadas variables que no se pueden determinar hasta qué están ocurriendo. En muchos ambientes hay reglas en las cuales no se puede realizar pruebas contra el sistema en producción, es importante recordar que el propósito de *chaos engineering* es descubrir el caos inherente en los sistemas, no provocarlo.

[5] Community, Chaos. *Principios de la Ingeniería del Caos*. https://principlesofchaos.org/. Consulta: 5 de marzo de 2021.

Si se sabe que algún experimento generará un resultado no deseado, entonces no se debería ejecutar ese experimento, dado que puede haber ciertas repercusiones donde el margen de error de la hipótesis será alto, y se debe estar preparado para tomar medidas en contra de estas situaciones.

1.5.4. Automatizar experimentos para que se ejecuten continuamente

Este principio reconoce la implicación de la carga de trabajo en sistemas complejos, donde indica que la automatización debe implementarse por 2 razones: Primero, para poder abarcar una gama más amplia de experimentos, en sistemas complejos, las condiciones que podrían contribuir a un incidente son muchas y no pueden ser planteadas, de hecho, ni siquiera pueden ser contadas porque son desconocidas. Esto significa que los humanos no pueden buscar de forma confiable una solución que contribuya a dichos factores en una cantidad razonable de tiempo. La automatización da un significado a la escala de investigación de vulnerabilidades que contribuyen a la salida de los sistemas. Segundo, para verificar los supuestos a través del tiempo, en cómo han cambiado de forma desconocida los componentes del sistema.[6]

1.5.5. Minimizar el radio de impacto

El último principio se agregó al grupo de los cuatro anteriores después de que el equipo formado en Netflix encontró que pudieron reducir considerablemente el riesgo de tráfico en producción por medio de la ejecución de experimentos, utilizando un rígido control de orquestación para comparar con la variable grupal, se pueden construir los experimentos de tal forma que el

[6] Community, Chaos. *Principios de la Ingeniería del Caos*. https://principlesofchaos.org/. Consulta: 5 de marzo de 2021.

margen de error de la hipótesis que el tráfico generado en producción para los clientes será mínimo.

Esto quiere decir, que es posible realizar experimentos controlados que no pongan en riesgo toda la estabilidad del sistema, únicamente para identificar fallas sin afectar al cliente final durante la ejecución de los experimentos.

1.6. Caos en la práctica

Basándose en Principles of Chaos Org, temas de caos la práctica difiere de la teoría establecida dado que no se puede predecir con exactitud qué pasará ante el cambio de variables en los diferentes entornos. Se puede tomar los principios de *chaos engineering* como una herramienta que facilita la generación de experimentos para descubrir debilidades en el sistema, donde los experimentos seguirán 4 pasos elementales según el artículo.

- Definir un estado saludable como una salida inicial del sistema, de forma medible, que nos indique el comportamiento normal del sistema.
- Generar una hipótesis en base a este estado normal que se manejara dentro de un grupo de control y experimental.
- Inducir variables que representan eventos del mundo real, como la inyección de latencia en la red que comunica los componentes, la falla de algunos servidores, desconfiguración de sistemas operativos, entre otros.
- Posteriormente, forzarse a refutar la hipótesis inicial buscando diferencias entre el estado normal definido sobre el sistema en su grupo de control contra el grupo experimental. Se puede considerar que entre más complicado sea provocar una falla en el estado normal, mayor confianza se tendrá sobre el funcionamiento del sistema. En el caso de descubrir alguna posible falla o vulnerabilidad, el objetivo será hacerlo más robusto

y mejorarlo antes de que dicho comportamiento se manifieste de forma general y a la vista de los usuarios finales.

1.7. ¿Puede soportar el servidor el ataque de un ejército de monos locos?

En base al artículo de Txema Rodríguez, indica que no existe ningún sistema que esté exento de tener fallas. No se puede asegurar que cualquier sistema que se ponga en producción no caiga en algunas horas. En base a las caídas a las que los usuarios de internet están acostumbrados, donde no pueden acceder a sus aplicaciones en algún determinado periodo de tiempo, *chaos engineering* plantea la idea de, en qué forma cualquier sistema distribuido se enfrenta a un ejército de monos que entrara a atacar cada componente de nuestro sistema y la interconexión de sus componentes, ¿Que podría pasar?,¿Somos capaces de continuar brindando el servicio con normalidad?¿Responderemos con algún error conocido o no podremos responder de ninguna forma?¿El sistema es lo suficientemente robusto para tolerar condiciones extremas?

1.8. ¿Dónde se puede agregar caos?

Tomando como punto de partida la idea de Casey Rosenthal y Nora Jones, ciertamente, es posible aplicar caos en diferentes partes de un sistema y en cada una reaccionara de forma distinta generando puntos donde las fallas suelen ser más “comunes” que en otras o por lo menos más fáciles de notar para un usuario; Algunas un poco difíciles para inducir caos de forma artificial, mas no imposible, y definen algunos puntos clave en donde se puede inducir fallos, siendo los siguientes:

1.8.1. Aplicación

Según Casey Rosenthal y Nora Jones, a nivel de aplicación es relativamente sencillo crear caos, basta con matar o suspender el proceso en ejecución de la aplicación, para verificar la tolerancia a fallos de esta y su capacidad de procesar la concurrencia. Esto sin necesidad de realizar un cambio a nivel de la programación de la aplicación ya creada, únicamente alterando su ejecución normal en su entorno.

1.8.2. CPU y memoria

Según Casey Rosenthal y Nora Jones, estos dos están relacionados directamente entre sí, al provocar una falla en alguno afectaría las operaciones del otro y su desempeño, una forma de poder aplicar caos en estos componentes es saturarlos o llevarlos por encima de su 80 % de capacidad por un cierto periodo de tiempo para verificar su comportamiento ante una sobrecarga de la cual el servidor no está exento de que ocurra, al utilizar sistemas distribuidos la saturación en un nodo no debería afectar al resto de forma visible, por lo que generar ciclos infinitos hasta llevarlo al máximo de utilización es un método rápido de verificar su falla temporal junto con el comportamiento de los demás componentes.

1.8.3. Red

Según Casey Rosenthal y Nora Jones, en ambientes en la nube, el correcto funcionamiento de la red es elemental para brindar y conectar servicios, es un punto idóneo para comenzar, desde algo tan simple como inyectar latencia en la red, simular una concurrencia de gran nivel para afectar el tráfico en la

comunicación de componentes o desconectar nodos dentro de la red para verificar la forma en la que el sistema se comporta ante estas situaciones.

1.8.4. Sistema operativo

Según Casey Rosenthal y Nora Jones, a nivel de sistema operativo se pueden generar diversos tipos de fallos que pueden o no afectar el sistema general y su funcionamiento como un todo. Una actualización errada puede generar fallas en todo el sistema, por medio de la entrada y salida de este es posible activar *triggers* que devuelvan errores sobre el sistema de archivos y generar inconsistencia en la estructura de este.

1.8.5. Infraestructura en la nube / *bare metal*

Según Casey Rosenthal y Nora Jones[7], dado que esto se maneja a nivel de los componentes "físicos" entre comillas dado que por ser en la nube no tenemos dichos componentes a nuestro alcance, pero en teoría son componentes físicos que podemos manipular, es la forma más sencilla de iniciar algunas pruebas, simulando la desconexión de componentes a nivel físico, apagar temporalmente algunos servidores de aplicaciones los cuales serían máquinas virtuales, disminuyendo abruptamente la cantidad de memoria en algunas máquinas virtuales o la desconexión de sus discos duros, sin importar el tipo. Cualquier falla que podamos abstraer de un componente físico la podemos traducir en una potencial amenaza para nuestro sistema, la cual media vez exista la posibilidad de que ocurra debemos considerar que así pasara.

[7] ROSENTHAL, Casey; JONES, Nora. *Chaos Engineering System Resiliency in Practice*. Inc, 2020. Consulta: 5 de marzo de 2021.

1.9. Ventajas y desventajas

Tomar la decisión de implementar experimentos de *chaos engineering* en un entorno funcional no es tarea fácil, además de tener en mente las posibles ventajas, siempre existen riesgos que se deben tomar en cuenta y aceptarlos o estar preparado para afrontarlos junto a una rápida solución, para poder aprovechar al máximo su potencial. Algunas de las ventajas y desventajas que pueden ayudar a la toma de decisiones se puede observar en la siguiente tabla.

Tabla I. **Ventajas y desventajas de *chaos engineering***

Ventajas	Desventajas
Ayuda a identificar múltiples vulnerabilidades.	Realizar un experimento no controlado puede afectar el sistema funcional.
Permite fortalecer el sistema.	Requiere un alto conocimiento de los componentes del sistema para experimentar con ellos.
Brinda un mejor entendimiento de los componentes de nuestro sistema y su interacción.	Es necesario tener una copia de seguridad de la arquitectura funcional en caso sea necesario aplicar un *rollback* de la misma por fallas persistentes.
Permite escalar rápido sin perder la confiabilidad del sistema.	
Ayuda a reducir la cantidad de incidentes.	
Se puede aplicar diversos escenarios y experimentos en conjunto para validar tantos escenarios como se considere necesario.	
Es posible aplicar experimentos sobre la arquitectura *bare metal* para evaluar su estabilidad y posibles puntos de quiebre.	
Ayuda a prevenir pérdidas económicas elevadas en el negocio a causa de fallas repentinas y reducir costos mediante la reducción del número de incidencias que afecten producción.	

Fuente: elaboración propia, empleando Microsoft Word.

1.10. Costos estimados

Para el uso de las herramientas, la gran mayoría de estas son de uso *open source*, para lo cual no se requiere una inversión por utilizarlas y existe suficiente información en los sitios oficiales de cada herramienta para aprender su uso, en la descripción posterior de cada herramienta se detalla en cada una si maneja algún costo por su uso o si es libre.

El verdadero gasto se refleja en el tiempo que una organización toma para el aprendizaje de las herramientas que encuentren útiles y en el manejo de sus métricas MTTR, MTBF, MTTF y demás gastos ocasionados por el tiempo que sus servicios permanezcan afectados por alguna falla o vulnerabilidad que no se consideró a tiempo, estos gastos generados no habían sido contemplados entre un presupuesto establecido y pueden incrementarse entre más tiempo persista la falla en un ambiente real.

Para esto se deberá contemplar el costo de las instancias en ejecución que presentaron fallas o de los diversos componentes que se mantienen en la nube, la pérdida monetaria que implica no poder brindar determinados servicios antes de la recuperación parcial o total del sistema.

2. SISTEMAS DISTRIBUIDOS

Tras el límite alcanzado por los sistemas monolíticos donde ya podían estar a la altura de las necesidades de crecimiento constante que manejan los sistemas modernos, se comenzó a fragmentar un solo sistema construido como un todo en sus diferentes componentes individuales de manera que estos nuevos sistemas pudieran trabajar a los ojos de los usuarios como un todo, aunque sus partes estuvieran dispersas en diferentes servidores.

2.1. Concepto

Estos serán los sistemas donde sus componentes de *software* y *hardware* estarán en distintas instancias conectadas por medio de una red, estos componentes se podrán comunicar y coordinar sus acciones transmitiendo datos entre sí por medio de algún protocolo establecido, dándole a los usuarios finales la sensación de estar utilizando un único servidor.

Al manejar los diferentes componentes de forma separada y no monolítica, es más sencillo agregar nuevos componentes o características para los usuarios dentro de esta arquitectura, permitiéndonos también escalar de forma más fácil los servicios en base a la demanda que se presente.

2.2. Características

A pesar de que los sistemas distribuidos no se manejan de una única forma, sino que cada diferente tipo de sistema distribuido estar orientado a resolver distintos problemas según sea necesario, pero se identifican

características en específico que todos estos sistemas comparten con la finalidad de resolver los diferentes problemas que puedan surgir.

2.2.1. Heterogeneidad

Los sistemas distribuidos pueden ser elaborados tomando como base distintas tecnologías, arquitecturas de redes, sistemas operativos, tipos de *hardware* utilizados para los servidores, lenguajes de programación, entre otros. Bajo estas circunstancias el sistema debe hacer uso de protocolos estandarizados y *middleware* para comunicar cada componente sin importar su forma de implementación dentro del sistema y poder enmascarar esta diferencia de componentes hacia los clientes finales, de esta forma ellos tendrán la impresión de estar utilizando un único sistema.

2.2.2. Escalabilidad

La escalabilidad es una característica de estos sistemas que nos permite aumentar o disminuir recursos a la capacidad que mantienen normalmente, es decir aumentar el número de instancias de una aplicación, entradas de red, entre otros. El objetivo de hacer esto es poder mantener un control de los costos de cada elemento y estar a la altura de la demanda de los servicios en cada momento, si la demanda es baja poder disminuir la capacidad o cantidad de elementos para no tener recursos perdidos y en caso la demanda incremente poder aumentar la capacidad y recursos del sistema para que los usuarios finales no identifiquen algún decaimiento en la calidad del servicio que están utilizando.

2.2.3. Tolerancia a fallos

La tolerancia a fallos nos dice que cualquier componente de un sistema podría fallar, un proceso, una computadora o toda la red pueden verse afectados independientemente de los demás y en un sistema distribuido se debe tener la capacidad de continuar funcionando y operando con relativa normalidad para todos los componentes que no fueron afectados.

Uno de los grandes beneficios de SOA es eso, que no todo el sistema fallará al mismo tiempo, en eso se basa la distribución de componentes, es decir mientras algún componente falle los demás continuarán funcionando, brindando una alta disponibilidad y reduciendo los tiempos fuera de servicio.

2.2.4. Concurrencia

Dado que estos sistemas brindan servicios que pueden ser utilizados por varios clientes a la vez, cada componente o servicio debe poder funcionar con una gran cantidad de accesos simultáneos, poder brindar su servicio y funcionar correctamente para los usuarios, tener la capacidad de sincronizar su información entre ellos para asegurar la consistencia de la información entre los servicios y funcionar con la misma capacidad en todo momento para satisfacer la alta demanda que pueda tener de parte de los usuarios finales.

2.2.5. Observabilidad

Esta característica es un pilar base de SOA, ya que nos dice que debemos recolectar información útil sobre el sistema para tener en la mira su comportamiento. Cuando ocurra algún fallo debemos recurrir a la información

recolectada y sus estadísticas para comprender de una manera más amplia por qué ocurrió un error y dónde ocurrió.

Para la implementación de observabilidad dentro de un sistema distribuido existen herramientas que podemos utilizar tanto de parte de los propios sistemas operativos como externas, se debe seleccionar la herramienta o gama de herramientas que mejor se adapte al entorno que ya se tiene.

2.2.6. Transparencia

La transparencia busca ocultar a los usuarios finales el hecho de que están manejando un sistema cuyos componentes están separados, de tal manera que para ellos se perciba como un todo en lugar de un grupo de componentes que pueden funcionar de manera independiente.

2.2.7. Seguridad

Dado que los componentes de estos sistemas están separados dentro de una red, estos podrían ser víctima de interferencias no deseadas y comprometer su información, por lo cual la seguridad debe encargarse de que todos estos envíos de paquetes tengan integridad y que no se degraden por ataques externos.

2.3. Arquitectura orientada a servicios

Es un estilo arquitectónico para construir soluciones empresariales basadas en servicios. Más específicamente, SOA se centra en la construcción independiente de servicios alineados al negocio que pueden ser combinados en una solución y procesos empresariales de mayor nivel. El verdadero valor de

SOA está en la reusabilidad de servicios y estos se combinan para crear procesos de negocio más flexibles y ágiles.[8]

Este paradigma busca que las unidades lógicas existan de forma autónoma, pero no aisladas unas de otras. Para esto es necesario encapsular las unidades lógicas de trabajo en segmentos de código más pequeños en base a su funcionalidad, interés en el negocio, entre otros.

Cada servicio puede ser utilizado por otros servicios según sea la necesidad o de forma individual, pero estos necesitan conocerse y entenderse, es decir que deben tener una forma estandarizada de cómo se van a comunicar con otros servicios ya sea internos o externos a la organización, los datos que son necesarios para transmitir información, el formato de envío y la forma en la que conocerán nuevos servicios, de esta forma hacer más sencilla la labor de implementar nuevas características de forma individual a sistemas ya existentes.

2.4. Arquitectura orientada a microservicios

Es un estilo de diseño arquitectónico de *software* que se enfoca en desarrollar aplicaciones como la unidad independiente más pequeña que se puede realizar. Al contrario de las arquitecturas convencionales monolíticas esta se divide en elementos pequeños capaces de funcionar en solitario, y a diferencia de la arquitectura orientada a servicios, busca generar servicios más pequeños separando un servicio en componentes más pequeños que pueden funcionar autónomamente comunicándose entre sí a través de mensajes y peticiones.

Esto ayuda a que cuando se generen nuevas características puedan ser desplegadas sin afectar a toda la aplicación o los demás componentes de esta,

[8] ROSEN, Mike. et al. *Applied SOA*. p. 70.

con la misma ventaja de que puede ser generado en condiciones distintas. A pesar de que manejar una arquitectura orientada a microservicios puede facilitarnos la capacidad de tener una alta disponibilidad y extensibilidad de características, todo trae nuevos retos, en el caso de los microservicios es la complejidad de su despliegue conforme la cantidad va aumentando levantar tantos servicios a mano es una tarea ardua y tardada por lo que es necesario comenzar a automatizar los despliegues de estos para hacerlo de una forma efectiva, al igual que la implementación de herramientas para su administración, coordinación y monitorización de su comportamiento.

2.5. Service mesh

Cuando se trabaja con cualquier tipo de arquitectura hay beneficios y retos implícitos, en el caso de los microservicios nacen diversos retos como la automatización de sus despliegues, la monitorización de cada microservicio desplegado y el tráfico de comunicación generado entre APIS de la red donde se han desplegado.

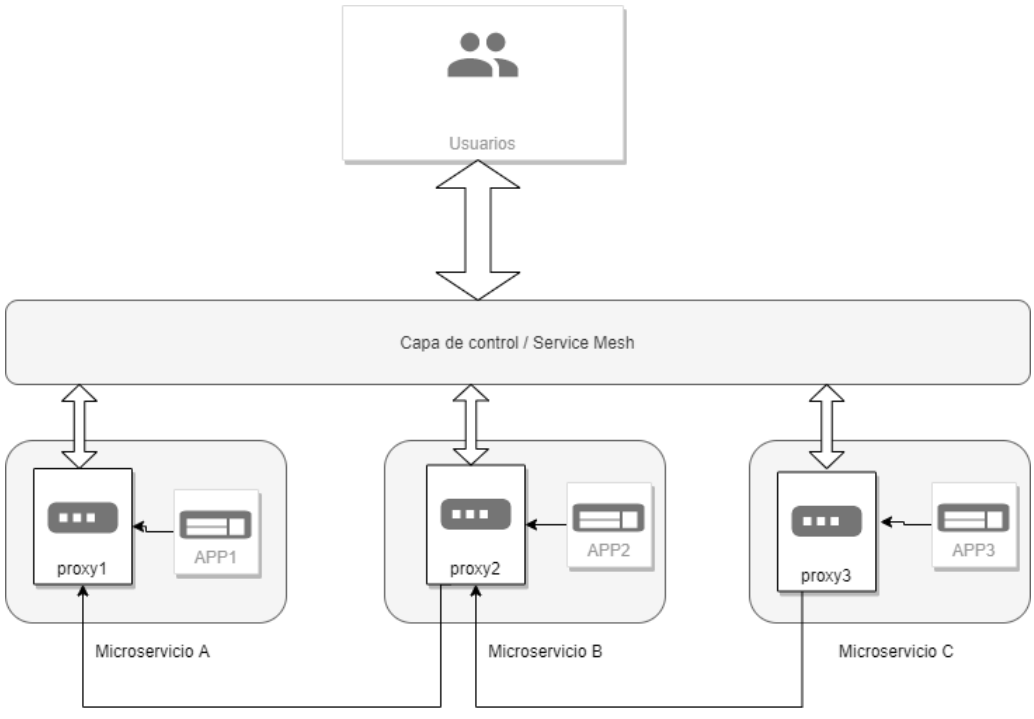
Al trabajar con microservicios, cada uno utiliza algún protocolo de alto nivel para su comunicación, pero no considera como se mueven los paquetes dentro de la red o como se gestiona. Se busca únicamente manejar de forma estandarizada las comunicaciones de estos microservicios y la comunicación de todos los componentes.

En este punto de estandarización entra *service mesh* al juego, como una capa dentro de la infraestructura que nos ayuda a que la comunicación sea flexible, rápida y confiable. Genera un equilibro de carga entre los componentes, autenticación, descubrimiento de servicios, monitoreo optimizado entre muchas otras características.

A grandes rasgos su funcionamiento busca que los microservicios no se comuniquen directamente, sino que se busca utilizar un *service mesh* como Linkerd para que la comunicación se realice a través de esta malla, utilizando una instancia de *proxy* como lo es Sidecar, en cada instancia del servicio. De esta forma se logra que la infraestructura se comunique al exterior a través de este *proxy* como una capa de control liberando la carga de los microservicios.

De esta forma se genera una capa de control con la cual los microservicios se comunican con su *proxy* únicamente, los *proxies* se pueden comunicar entre ellos para el manejo de la comunicación interna y la capa de control o el *service mesh* será el que se encargue de gestionar cada *proxy* de forma unificada.

Figura 1. **Descripción *service mesh***



Fuente: elaboración propia, empleando Draw.io.

2.5.1. Historia

La historia de *service mesh* es orientada al comportamiento de las arquitecturas y los retos que surgen con cada nuevo modelo arquitectónico. Inicialmente con las arquitecturas monolíticas era un control relativamente sencillo, se instalaba todo lo necesario en una computadora que se brindaba el servicio, pero era complicado de escalar, actualizar y propenso completamente a fallos, si fallaba esa única instancia donde estaba toda la aplicación, hacía imposible utilizar cualquier parte del servicio, aunque el error fuera en un solo componente.

Esto dio origen a los sistemas distribuidos donde cada aplicación se divide en componentes grandes o pequeños ubicados en diferentes computadoras, solventando el problema de fallas totales, ahora si un componente fallaba no afecta todo el sistema, sino que únicamente dicho componente, pero con cada reducción abstracta que se realizaba para llevar cada servicio a su mínima expresión funcional (microservicios) la cantidad de computadoras donde se ejecutaban aumentaban considerablemente generando un nuevo problema, el tráfico generado en desorden para la comunicación de estas computadoras.

Es allí donde por fin surge *service mesh*, como una solución al tráfico en aumento y desordenado generado por sus predecesores, buscando traer un orden abierto a estas comunicaciones y brindando la opción de poder monitorear de mejor forma esta comunicación junto con su comportamiento con las instancias en ejecución.

2.5.2. El desafío de ir más rápido

Se han definido diversos retos con el manejo de microservicios conforme se adopta por diferentes equipos. Tomando como el problema principal la comunicación de los microservicios y la red confiable en la que se manejan, muchas empresas han lanzado *frameworks* propios para ayudarse a tener una red confiable, pero el uso de contenedores Linux y Kubernetes/OpenShift ha sido fundamental para permitir a los equipos DevOps alcanzar mayores velocidades enfocándose en una imagen inmutable que se maneja a través de un pipeline automatizado.

El cómo cada equipo maneja sus pipelines es independiente del lenguaje o *framework* que este en el contenedor, las herramientas y la infraestructura utilizada para implementar y administrar servicios distintos están madurando, pero aún faltan capacidades similares cuando se habla de cómo interactúan estos servicios entre sí. Aquí es donde se debe buscar construir *software*, entregarlo y comunicarlo más rápido que nunca.

2.5.3. Observabilidad

Uno de los retos con la administración de los microservicios es intentar entender las relaciones individuales entre los componentes sobre todo el sistema. Una sola transacción de algún usuario puede atravesar por múltiples microservicios ejecutándose en diferentes instancias y descubrir donde se generan cuellos de botella o donde ocurre un error brinda información muy importante, obteniendo esta información a través de trazas y métricas.

2.5.4. Resiliencia

La resiliencia es la habilidad de sobreponerse y adaptarse a alguna situación adversa, teniendo resultados positivos ante esta.

Todos nuestros servicios y aplicaciones se comunican sobre una red poco confiable. Los sistemas previamente eran susceptibles a fallas en cascada cuando ocurría una falla en la red, no se debe asumir que una dependencia remota de los microservicios que accede a través de una red tendrá garantizado que responda algo valido o si quiera si responderá, para eso hay que tomar en cuenta ciertas características que se deben aplicar para que nuestro sistema posea resiliencia:

- Balanceo de carga del lado del cliente: se recomienda el balanceo de carga de Kubernetes *out-of-the-box*.
- *Timeout*: se esperará X segundos una respuesta o se cede en esta.
- *Retry*: si algún *pod* devuelve un error, intentar hacia otro *pod*.
- *Simple circuit breaker*: en vez de presionar al servicio degradado, abrir un circuito y rechazar nuevas solicitudes.
- *Pool ejection*: brinda la eliminación automática de *pods* propensos a errores del grupo de equilibrio de carga.

2.5.5. RPC

La llamada a procedimientos remotos es una herramienta que nos apoya en la comunicación generada entre procesos. Este proceso consiste en el envío de parámetros y el retorno de un valor de alguna función, aunque se limita a una sola llamada en la práctica se generan múltiples solicitudes en paralelo.

2.5.6. Data plane

Esta será la capa del *Service mesh* que se encargará de mover los datos de cada solicitud en el servicio a través de la red en tiempo real. Esta capa se basa en la implementación de *proxies* interne conectados que utiliza cada microservicio.

El *data plane* está construido de tal forma que intercepta todo el tráfico entrante y saliente, los microservicios pueden utilizar esta característica del *data plane* para invocar *endpoints* HTTP remotos para descubrir servicios.

2.5.6.1. Envoy Proxy

Este es un *proxy* que se maneja en la capa 7 del modelo OSI desarrollado por Lyft, es capaz de manejar millones de solicitudes por segundo y fue escrito en C++. Es ligero y de muy alto rendimiento para el procesamiento de solicitudes concurrentes y una de sus grandes ventajas es el balanceo de carga para HTTP1 o HTTP2, junto con gRPC.

Tiene la capacidad de recolectar métricas a nivel de solicitudes, realizar trazas, brindar conocimiento de servicios, inyectar fallas, entre otros.

2.5.7. Control plane

Esta capa del *service mesh* actúa romo un enrutador, se encarga de crear tablas de ruteo o el uso de varios protocolos para identificar caminos en la red y almacena estos caminos. Esta información la obtiene el *data plane* para modificar el comportamiento del servicio.

2.5.8. Adaptadores

Este es un patrón arquitectónico donde el adaptador del contenedor es utilizado para modificar la interfaz de la aplicación contenerizada para conformar alguna interfaz predefinida que es esperada por todas las aplicaciones. Busca llevar un mejor control la heterogeneidad del sistema por medio del control de acceso, un monitoreo efectivo y la verificación de la salud de estos.

2.5.8.1. Monitoreo

Se busca manejar el monitoreo de un sistema de forma unificada, pero con los sistemas distribuidos debido a la amplia gama de código que se ha implementado en distintos lugares, ocurre que se genera una amplia gama de interfaces distintas de monitoreo que generan la necesidad de integrarse en un sistema más fácil de entender. Afortunadamente muchas soluciones actuales de monitoreo comprenden esta necesidad y han implementado distintos plugin que se adaptan a las necesidades.

Aplicando el patrón Adaptador para monitoreo seremos capaces de ver que el contenedor de aplicación es únicamente la aplicación que deseamos monitorear sin tanta complejidad. Desacoplando el sistema de esta forma se hace una tarea más comprensible de brindar mantenimiento posteriormente.

2.5.8.2. Acceso

Así como con el monitoreo hay una gran cantidad de heterogeneidad en como los sistemas acceden a la información. Los sistemas deben separar sus accesos en diferentes niveles y cada nivel con diferentes permisos. Agregando

más complejidad, la información normalmente está estructurada de distintas formas.

El patrón Adaptador brinda un diseño modular y reutilizable para todas estas situaciones. Mientras que las aplicaciones acceden a través de un archivo, el adaptador lo redirecciona, de esta forma cuando diferentes aplicaciones acceden a la información con diferentes formatos, el adaptador transforma esta data en una representación estructurada que puede ser consumida. En pocas palabras el adaptador toma las aplicaciones heterogéneas y crea una interfaz homogénea en común para el acceso.[9]

2.5.8.3. Agregando un monitor de salud

Suponga que se desea manejar métricas de salud en un contenedor, un orquestador nos permite agregar esto por medio de la ejecución de procesos y escucha de algunos puertos en específico. ¿Qué sucedería si deseáramos estadísticas de estado más específicas? Kubernetes nos permite escribir *scripts* complejos para ejecutar diferentes diagnósticos, pero ¿Dónde almacenaríamos dichos *scripts* y cómo podríamos versionarlo?

2.5.9. El patrón Sidecar

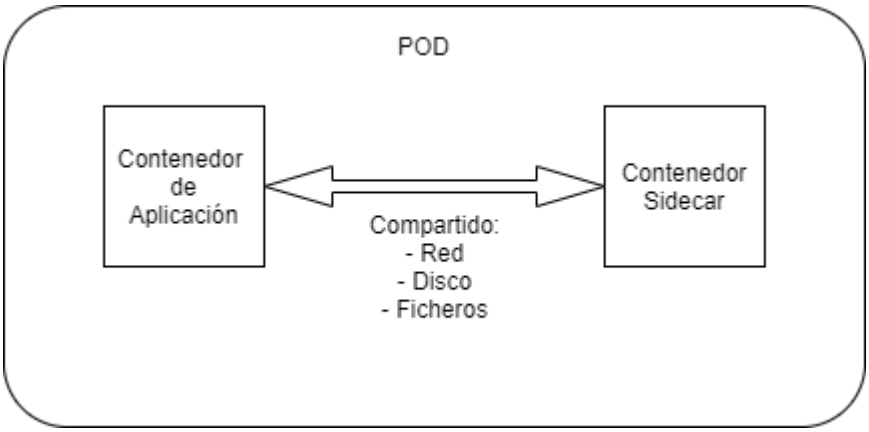
La respuesta a esto es utilizar el patrón Adaptador, se puede agregar un adaptador que contenga únicamente *scripts* para verificar el estado del sistema o de contenedores específicos y en caso de que algún *script* falle tomar medidas inmediatas. Adicionalmente al contenedor de la aplicación se agrega un

[9] BURNS, Brendan. *Designing Distributed Systems*. p. 34

contenedor Sidecar, cuyo rol es aumentar y mejorar el contenedor de la aplicación, normalmente sin el conocimiento de este.[10]

En pocas palabras el contenedor sidecar se usa para agregar funcionalidad a un contendor que de otra forma seria difícil de mejorar. Estos contenedores se programan en la misma maquina en un grupo, como el *pod* en Kubernetes. Al agregarse en la misma instancia ambos contenedores comparten recursos como sus sistemas de archivos, nombre de *host*, red, entre otros, es decir que el Sidecar puede contener al *proxy* del microservicio recibiendo las solicitudes para el servicio principal capturándolo y reenviándolo como un intermedio.

Figura 2. **Ejemplo patrón Sidecar**



Fuente: elaboración propia, empleando Draw.io.

[10] BURNS, Brendan. *Designing Distributed Systems*. 12 p.

2.5.9.1. Aplicación modular con contenedores

El uso de Sidecar no se limita únicamente a ser adaptado en aplicaciones legado cuando ya no se desea modificar el código fuente original, a pesar de que es una gran ventaja, también es utilizado para la modularidad y reutilización de dichos componentes como un Sidecar. Es decir, si se desea incrementar las funcionalidades de una aplicación ya existentes subdividiendo esta, en vez de manipular el código original que ya está funcionando se puede agregar esto en uno o varios Sidecar por medio de un sistema de orquestación para tener un grupo de funcionalidades consistentes ejecutados y disponibles en la arquitectura.

2.5.10. Casos de uso

A continuación, se detallan algunas de las actividades o casos en los cuales se puede hacer uso de *service mesh*, para aprovechar al máximo sus capacidades descritas anteriormente.

2.5.10.1. Reliability

Brinda una garantía de que funcionara bajo ciertas condiciones, por ejemplo, para el manejo de fallas, errores y la administración de peticiones *retry* de accesos para evitar pérdidas económicas en los diferentes servicios.

2.5.10.2. Service discovery

En *service mesh* dado que se brinda un nivel alto de disponibilidad, da un nivel de servicio de visibilidad y telemetría que ayuda a que se genere un catálogo de información de servicios y realizar un análisis de dependencia preciso.

2.5.10.3. Migración de sistemas

Al realizar una migración completa o parcial de un sistema, se puede subdividir en sus componentes y agregarlos de forma periódica, por medio de la redirección del tráfico desviar las solicitudes respectivas de cada componente entre el sistema antiguo y el nuevo, de esta forma los usuarios finales no notarán ningún cambio de su lado ni se verán afectados por un *downtime* de la aplicación durante una migración completa.

2.5.10.4. Traffic governance

Dado que el funcionamiento de *service mesh* toma peso en el uso de la red, se puede generar reglas aplicadas al tráfico de esta, sin necesidad de modificar ninguna de las aplicaciones en ejecución dentro del sistema, tanto para el ingreso y egreso de paquetes; así también poder llevar el control del tráfico que atraviesa dentro de nuestro sistema.

2.5.10.5. Secure service-to-service communications

Service mesh permite realizar el proceso de comunicación de forma segura en la red utilizando TLS, este es un protocolo que encripta la información que se envía a través de internet para prevenir que atacantes externos puedan ver qué información se está transmitiendo, asegurando el envió de paquetes dentro de la red de servicios que trabajan directa o indirectamente.

2.6. Ventajas

Los sistemas distribuidos proveen una amplia cantidad de ventajas en comparación de los sistemas monolíticos, algunas de estas ventajas se listan a continuación.

- Es más barato poder agregar o reducir servidores de forma temporal en base a la demanda que adquirir recursos permanentemente, aunque estos estén sin uso.
- Se libera la carga de todos los procesos de una sola instancia.
- Se puede distribuir el tráfico y la carga de trabajo entre diferentes instancias y localidades.
- Es menos propenso a fallas completas, dado que si un componente falla no afectara la disponibilidad del resto del sistema.
- Se maneja mediante un crecimiento modular.

2.7. Desventajas

A pesar de las numerosas ventajas que nos proveen los sistemas distribuidos, manejarlos e implementarlos conllevan ciertos riesgos que hay que tener en cuenta.

- El diseño e implementación de estos sistemas requiere de un mayor conocimiento técnico sobre las dependencias y relaciones de los servicios.
- Dado que los componentes se distribuyen en una red, esta puede sufrir interferencias de terceros.
- La administración de estos suele ser más compleja.

2.8. *Cloud native*

Según la definición oficial de la CNCF, las tecnologías *cloud native* empoderan a las organizaciones para construir y correr aplicaciones escalables en ambientes dinámicos modernos, como lo son hoy las nubes públicas, privadas o hibridas. Temas como contenedores, *service mesh*, microservicios, infraestructura inmutable y APIS, declarativas son ejemplos de este enfoque.

Estas técnicas permiten crear sistemas de bajo acoplamiento que son resilientes, administrables y observables. Combinado con técnicas de automatización robusta permite a los ingenieros realizar cambios de alto impacto de forma frecuente y predecible con un mínimo esfuerzo.

Se busca impulsar la adopción de este paradigma mediante el fomento y mantenimiento de un ecosistema de proyectos de código abierto y neutro con respecto a los proveedores. Democratizando los patrones modernos para que estas innovaciones sean accesibles.

2.9. CNCF

La *cloud native* Computing Foundation por sus siglas en inglés, es una organización derivada de Linux Foundation que busca el avance de las tecnologías de contenedores y alinear a la industria alrededor de esta evolución.

CNCF maneja un grupo de proyectos tecnológicos que son catalogados según su nivel de madurez como sandbox, en incubadora y graduados. Estos proyectos son las tecnologías que forman parte de este paradigma, probados para un manejo avanzado y aprovecha de mejor manera la implementación de *cloud native*.

2.10. Características

Los sistemas *cloud native* poseen ciertas cualidades que lo hacen diferente a los sistemas convencionales *on premises* o híbridos donde únicamente se cuenta con algunos servicios en la nube, algunas de estas características se detallan en el siguiente listado.

- Bajo acoplamiento: al estar separado en microservicios o *service mesh*, el realizar cambios en un componente afectara lo menos posible a los demás.
- Administrable: se tiene un riguroso control sobre los componentes para poder tomar distintas acciones en cualquier punto según sea necesario.
- Aplicaciones escalables: es posible aumentar o disminuir las capacidades y recursos de cada componente dentro del sistema en base a la demanda de cada uno; de la misma forma agregar o quitar componentes según la orientación del negocio.
- Resilientes: es la capacidad de un sistema para poderse adaptar y continuar funcionando incluso en situaciones adversas, gestionando la situación de tal forma que su funcionamiento se afecte de la menor forma posible.
- Observables: mediante la aplicación de herramientas de monitoreo es posible llevar un estatus en vivo del estado de nuestro sistema implementando las métricas que se consideren de mayor importancia, latencia, vigilancia de paquetes, *health-checks* de instancias, entre otros.

2.11. Tecnologías y practicas utilizadas

Para ser considerado *cloud native* un sistema debe hacer uso de ciertas tecnologías y prácticas que lo enfocan a manejarse completamente en la nube,

sin manejar nada de forma local, algunas de estas tecnologías se detallan a continuación.

2.11.1. Plataformas en la nube

Siendo uno de sus pilares debido a la facilidad de configuración y adquisición de recursos para poder desplegar distintos servicios, las plataformas en la nube nos brincan la posibilidad de montar servidores a costos más bajos que adquiriéndolos *on premises*. Se tiene la capacidad de tener el control y administración de los recursos que se están utilizando, así como herramientas predeterminadas e integración en red para alta disponibilidad independientemente de la plataforma en la cual se manejen los servicios.

2.11.2. Contenedores

Un contenedor en el ámbito informático será una pequeña virtualización de un sistema completo que tendrá lo necesario para trabajar. Es decir que en un paquete se podrá almacenar todo lo necesario para la ejecución de alguna aplicación y podrá ser comprimido y reutilizado desde cualquier lugar, esto significa que en vez de realizar todas las instalaciones necesarias en repetidas ocasiones para un contenedor únicamente se puede tomar su imagen con todo lo necesario ya instalado y funcional para ejecutarlo en diferentes instancias.

La herramienta para contenedores más utilizada actualmente es Docker, la cual es de código abierto para su uso.

2.11.3. Kubernetes

Es una plataforma de código abierto que permite la administración de cargas de trabajo y servicios facilitando la automatización y configuración declarativa. Esta plataforma pertenece a Google, como resultado de la experiencia que generaron al ejecutar aplicaciones en producción de una forma masiva por alrededor de una década y media.

Kubernetes se enfoca en la administración de contenedores orquestando su infraestructura, redes y almacenamiento para hacer más fácil la labor de crear una infraestructura compleja.

2.11.4. CI/CD

La integración continua es una práctica utilizada en desarrollo de software en el cual se pueden unificar los cambios realizados en algún proyecto de forma periódica dentro de un repositorio central. Dentro de este se pueden realizar pruebas, identificar fallos y repararlos de forma rápida.

Mientras que la entrega continua sería el paso posterior a la integración. Esto garantiza que se podrá integrar el código nuevo dentro de un ambiente de producción y entregar actualizaciones constantes con el código ya funcional.

Su mayor beneficio es en la automatización dado que permite automatizar todas las tareas, desde la unión del código, por medio de la realización de todas las pruebas hasta el despliegue automático del código dentro de nuestra arquitectura.

3. DEFINICION PREVIA A LA GENERACION DE CAOS

Antes de poder realizar experimentos y llevar el sistema al límite es necesario considerar ciertos puntos ¿Qué herramienta se adapta mejor a los experimentos que se quiere ejecutar?, ¿Mi arquitectura tiene lo necesario para aplicar estos experimentos?, ¿Conozco alguna secuencia de pasos para poder realizar dichos experimentos?, para lo cual en los siguientes puntos se establece la pregunta a estas interrogantes y se evalúa paso a paso los puntos necesarios antes de poder proceder con la ejecución de los experimentos.

3.1. Evaluación de herramientas para crear caos

Actualmente hay una gran variedad de herramientas que nos permiten generar caos en diferentes tipos de ambientes, cada herramienta con un fin u objetivo específico, aunque algunas abarcan más de un área de impacto que se puede aplicar.

3.1.1. Linkerd

Linkerd es un *proxy* de uso *open source* que se creó para ser utilizado como un *service mesh*, es decir que esta plataforma nos permite implementar un *service mesh* utilizándolo, interconecta distintos componentes de los sistemas distribuidos, siendo esta herramienta pionera en el campo desde antes que se tomara como concepto los *service mesh*.

Nos brinda la funcionalidad de controlar y monitorear cada componente y servicio desplegado, balanceo de carga, *circuit breaking* y *retry* para peticiones fallidas, conexiones TLS a nivel de la seguridad de la red, enrutamiento dinámico de los componentes, monitoreo de métricas y una de sus inclusiones para pruebas es poder simular altas cargas de peticiones a la red de forma introductoria para la aplicación de *chaos engineering*.

Tabla II. **Características de Linkerd**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
open source	*graduated*	Red	• *fault Injection*
		service mesh	• Permite la implementación de malla de servicios con su sistema.

Fuente: elaboración propia, empleando Microsoft Word.

3.1.2. Chaos Mesh

Esta es una herramienta *open source* para plataformas *cloud native* específicamente, permite la orquestación de caos en entornos elaborados con Kubernetes, donde su funcionamiento gira entorno a la inyección de fallas en estos entornos incluyendo fallas en *pods*, fallas de red, sistemas de archivos e incluso se pueden inducir fallas al Kernel de los contenedores.

Tabla III. **Características de *chaos mesh***

Licencia	Estado del proyecto en CNCF	Experimentos soportados	Detalle de experimentos
Open source	*Incubating*	Pod	• *Pod failure* • *Pod kill* • *Container kill*
		Red	• *Network partition action* • *Network loss* • *Network delay* • *Network duplicate* • *Network corrupt* • *Network bandwidth action*
		Stress	• Utilización de CPU
		Tiempo	• Modificar el reloj en *pods*
		I/O	• *Latency* • *Faults* • *AttrOverride* • *Mistake*
		Kernel	• Desconfigurar sistema de archivos
		DNS	• Genera un DNS propio para inducir fallas
		JVM	• Excepciones personalizadas • Modifica valores
		AWS Chaos	• *EC2 stop* • *EC2 restart* • *Detach volume*
		Azure Faults	• *VM stop* • *VM restart* • *Disk detach*
		GCP Faults	• *Node stop* • *Node reset* • *Disk loss*
		HTTP Faults	• Abortar conexiones • Retraso

Fuente: elaboración propia, empleando Microsoft Word.

3.1.3. Chaos Monkey

Esta es una herramienta *open source* para plataformas *cloud native* específicamente, permite la orquestación de caos en entornos elaborados con Kubernetes, donde su funcionamiento gira entorno a la inyección de fallas en estos entornos incluyendo fallas en *pods*, fallas de red, sistemas de archivos e incluso se pueden inducir fallas al Kernel de los contenedores. La herramienta posteriormente la liberaron para ser *open source*.

Su funcionamiento está orientado principalmente a provocar fallas aleatorias dentro de las instancias de la nube, pero se limita a provocar fallas a este nivel únicamente y no posee capacidad de recuperación, es decir, si un experimento se sale de control se debe detener de forma manual y restablecer los servicios de la misma manera.

Tabla IV. **Características de Chaos Monkey**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	N/A	Instancias	• Termina instancias y contenedores de forma aleatoria

Fuente: elaboración propia, empleando Microsoft Word.

3.1.4. Litmus

Es un grupo de herramientas orientadas a *chaos engineering* aplicado a infraestructuras *cloud native*. El objetivo de esta herramienta es apoyar a identificar vulnerabilidades dentro de nuestra arquitectura con Kubernetes orquestando el caos dentro de esta, permitiendo realizar experimentos de forma fácil en producción. El proyecto Litmus es *open source*, su código y uso se encuentran en su sitio oficial así también como en su repositorio en Github.

Tabla V. **Características de Litmus**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	*Incubating*	Genérico	• *Pod delete* • *Container kill* • *Pod network latency* • *Pod network loss* • *Pod CPU hog* • *Pod memory hog* • *Pod network corruption* • Docker *service kill* • *Pod network duplication* • *Disk fill* • *Disk loss* • *Kubelet service kill* • *Pod IO stress* • *Node memory hog* • *Node restart* • *Pod http latency* • *Pod autoscaler*
		AWS	• EC2 *terminate* • EBS *loss* • AWS AZ *chaos*

Continuación tabla V.

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	*Incubating*	Cassandra	• *Pod delete*
		Kafka	• *Broker pod failure* • *Bróker disk failure*
		GCP	• *Instance stop* • *Disk loss*
		Azure	• *Vm instance stop* • *Vm disk loss*

Fuente: elaboración propia, empleando Microsoft Word.

3.1.5. Latency Monkey

Esta aplicación también es parte del grupo de herramientas desarrolladas por Netflix, aunque posterior a Chaos Monkey. Se enfoca en causar retrasos en las comunicaciones cliente-servidor de las aplicaciones *RESTfull*, su única función es generar *delay* y fallas dentro de la red y sus dependencias para verificar de forma responderá el sistema. Al igual que su predecesor esta herramienta también es *open source*.

Tabla VI. **Características de Latency Monkey**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	N/A	Red	• *Delays* • Latencia

Fuente: elaboración propia, empleando Microsoft Word.

3.1.6. Gremlin Inc.

Es un grupo de herramientas que permite realizar experimentos y ataques a infraestructuras basadas en Kubernetes y la nube, teniendo como objetivo generar fallas en aplicaciones, librerías e interfaces de usuario. Esta gama de herramientas está bajo una licencia de pago para poderse utilizar.

Tabla VII. **Características de Gremlin Inc.**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
De Pago	N/A	Gremlins de recursos	• Alta carga en CPU • Bloquear cantidades de RAM • Agregar alta carga de I/O al disco duro • Generar archivos para llenar el disco duro
		Gremlins de estado	• Apagar instancias • Cambiar hora del sistema • Mata procesos
		Gremlins de red	• *Blackhole* • Latencia • Perdida de paquetes • Bloquear DNS

Fuente: elaboración propia, empleando Microsoft Word.

3.1.7. Chaos Toolkit

Este es un proyecto *open source* impulsado por las comunidades, diversificando su objetivo para liberar herramientas que permitan aplicar diferentes tipos de experimentos en los entornos en la nube. Es compatible con Kubernetes como las herramientas anteriores, pero también se puede aplicar directamente sobre instancias de diferentes proveedores de servicios en la nube.

Tabla VIII. **Características de Chaos Toolkit**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open Source	N/A	genérico	• Permite crear experimentos personalizados con código

Fuente: elaboración propia, empleando Microsoft Word.

3.1.8. Toxiproxy

Es un *framework* que permite la simulación de condiciones en red dado que es el objetivo principal con el que se creó. Toxiproxy es utilizado cuando se necesita realizar pruebas en aplicaciones que no contienen un punto de fallo único siendo este también *open source*.

Consiste en 2 partes. la primera es un *proxy* TCP escrito en lenguaje GO y el cliente que comunica con el *proxy* a través de peticiones http.

Tabla IX. **Características de Toxiproxy**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	N/A	Red	• Latencia • Degradación de ancho de banda • *Delays* • *Jitter* • *Slice* TCP *data* • Cierre aleatorio de conexiones

Fuente: elaboración propia, empleando Microsoft Word.

3.1.9. Kubeinvaders

Esta herramienta es estéticamente distinta a las descritas anteriormente, las cuales se basan en realizar experimentos formales, pero, Kubeinvaders toma la analogía del juego de *arcade* Space Invaders. Busca realizar los experimentos de chaos engineereing de una forma divertida aplicada a realizar pruebas sobre los clústeres de Kubernetes para medir su resiliencia.

La forma de uso general de la herramienta es similar a un simulador del juego de arcade, donde las naves enemigas serán representadas por *pods* a los cuales se destruyen. La herramienta es *open source* teniendo su mantenimiento por una pequeña comunidad de personas apasionadas por los experimentos de *chaos engineering* y los juegos de antaño.

Tabla X. **Características de Kubeinvaders**

Licencia	Estado del Proyecto en CNCF	Experimentos soportados	Detalle de Experimentos
Open source	N/A	Kubernetes	• Eliminación espontanea de *pods*/clústeres

Fuente: elaboración propia, empleando Microsoft Word.

3.2. Comparativa de herramientas para caos

Escoger una herramienta para poder llevar a cabo experimentos de *chaos engineering* puede parecer tarea sencilla, pero hay que tomar en cuenta que cada herramienta está orientada a uno o más entornos y experimentos que puede realizar, costos de licenciamiento en algunos casos, entre otros. por lo cual en la siguiente tabla se detallan ciertas características para la selección de herramientas.

Tabla XI. **Comparativa de herramientas para *chaos engineering***

Características	**Linkerd**	**Chaos Mesh**	**Chaos Monkey**	**Litmus**	**Latency Monkey**	**Gremlin Inc**	**Chaos Toolkit**	**Toxiproxy**	**Kubeinvaders**
Licencia de pago						X			
Open source	X	X	X	X	X		X	X	X
Pod		X		X					X
Red	X	X		X	X	X		X	
Estrés		X		X		X			
Tiempo		X				X			
I/O		X		X		X			
Kernel		X				X			
DNS		X				X			
Instancias			X			X			
AWS		X		X					
GCP		X		X					
Azure		X		X					
Experimentos genéricos				X			X		
Trabaja con CNCF	X	X		X					

Fuente: elaboración propia, empleando Microsoft Word.

3.3. Identificación de posibles costos

En resumen, en lo definido por la teoría y las diferentes herramientas mostradas anteriormente, la mayoría de las herramientas para la implementación de *chaos engineering* son *open source* y poseen una gama extensa de experimentos para poder realizar por lo que implementar esta disciplina no representa gastos adicionales por la adopción de alguna de las herramientas descritas.

En el caso de escoger una herramienta de pago, el costo de uso será variable dependiendo de los precios impuestos por estas herramientas y la forma de su uso.

3.4. Ciclo de aplicación de *chaos engineering*

Para la aplicación y ejecución de experimentos de caos, es necesario seguir un flujo básico que ayudara a definir lo que queremos lograr con cada experimento, este flujo se encuentra distribuido en los diferentes pasos que se muestran en esta guía de forma implícita y se muestran a continuación.

Figura 3. **Ciclo de aplicación de *chaos engineering***

Fuente: elaboración propia, empleando Microsoft PowerPoint.

3.5. ¿Qué *framework* se necesita?

Para los fines de la experimentación del presente trabajo es necesario utilizar un marco de trabajo que este contenerizado sobre alguna plataforma de *cloud computing* enfocándose hacia donde se dirige la demanda de implementación actualmente, el entorno se inyectara con un *service mesh* para su observabilidad, seguridad y administración durante la ejecución de los experimentos.

Los experimentos serán realizados con una herramienta que genera un *service mesh* y Chaos Mesh la cual fue seleccionada para la ejecución de las pruebas de caos sobre el *framework* establecido debido a la amplia cantidad de experimentos que permite generar y verificar desde su *dashboard* administrativo.

Las herramientas seleccionadas para establecer un marco sobre el cual poder experimentar se detallan a continuación.

3.5.1. Herramientas utilizadas

A continuación, se detallan las herramientas utilizadas para los experimentos en el presente trabajo.

- Ubuntu 20.04: se utilizó esta versión del sistema operativo Linux con interfaz gráfica instalado y configurado localmente, esto debido a la facilidad que posee el sistema para manejarse a nivel de una línea de comandos en conjunto con su interfaz gráfica y su alto nivel de personalización sin restricciones.

- Google Cloud: se utilizó la plataforma Google Cloud para establecer la arquitectura en la nube completamente debido a que presenta los costos más bajos en comparación a sus 2 equivalentes, además de brindar una administración nativa con Kubernetes.

- Docker: es el proyecto *open source* que nos permitirá el despliegue de los diferentes componentes de la arquitectura contenerizados en diferentes imágenes.

- Docker Hub: es la biblioteca de imágenes para contenedores más grande del mundo, la cual se estará utilizando para almacenar las imágenes definidas con Docker, el enlace mediante el cual se pueden ver y clonar las imágenes está en los anexos del documento.

- Kubernetes: esta es la plataforma de contenedores que se utilizó para orquestar el despliegue y administración de la arquitectura definida de una forma rápida. Las versiones utilizadas para el cliente fueron 1.23.0 y para la versión de servidor 1.22.0.

- Linkerd: es el servicio *open source* utilizado para desplegar la arquitectura como *service mesh* dentro de Kubernetes, para administrar la observabilidad, *sidecar proxy* y seguridad de la arquitectura. Además de proveer un *dashboard* sobre el cual se puede monitorear el Desarrollo de la arquitectura y trafico fallido dentro de la misma, instalado en su versión estable 2.11.2

- Chaos Mesh: es la herramienta seleccionada para la ejecución de los experimentos de caos que se aplicaron sobre la arquitectura, instalado en su versión estable 2.2.0

- Github: es el repositorio de código utilizado, en el enlace ubicado en los apéndices se podrá acceder, en el estará todo el código utilizado para los experimentos y como control de versiones se utiliza el sistema Git en su versión 2.25.1.

3.6. Definir una arquitectura

Con fines de experimentación se definió una arquitectura que se puede clonar del repositorio ubicado en los anexos, esta arquitectura esta está escrita sobre código Yaml para que los usuarios únicamente ejecuten la arquitectura y los experimentos con los pasos posteriores.

La arquitectura comprende en 3 bloques principales funcionando sobre un clúster de Kubernetes creado dentro de Google Cloud, para la ejecución de la arquitectura contenerizada y la ejecución remota de cada uno de los experimentos descritos.

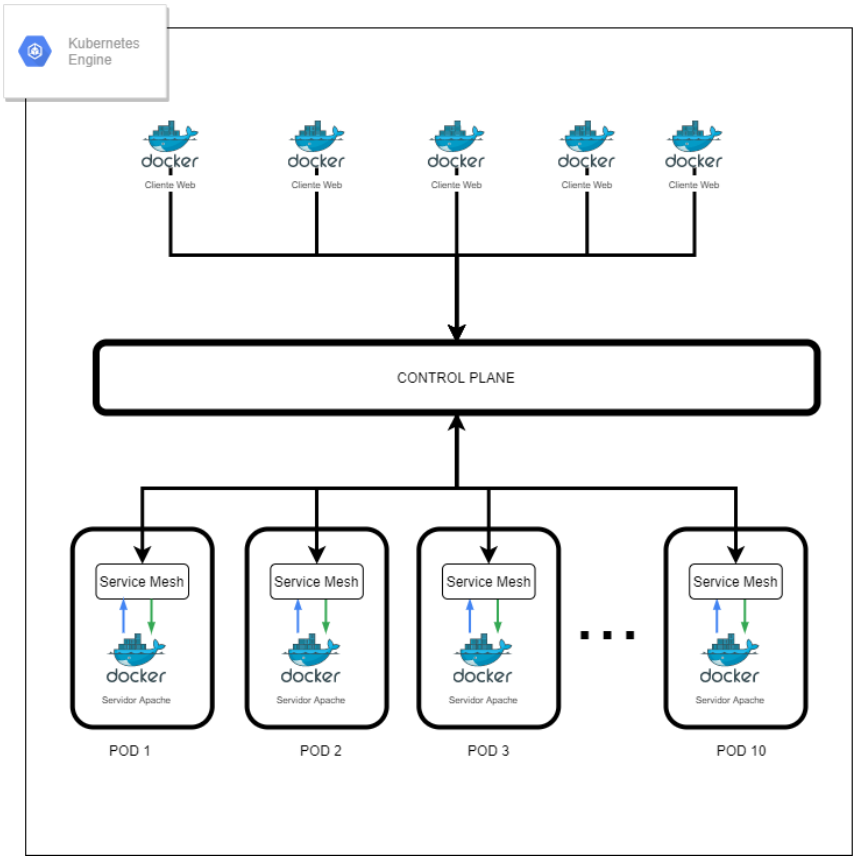
El primer bloque consiste un cliente que estará enviando solicitudes y generando tráfico de forma constante hacia servidor. Este cliente está configurado para generar 5 réplicas con la misma funcionalidad dentro de un contenedor.

El segundo bloque será el servidor contenerizado con un servicio de Apache que estará únicamente recibiendo las peticiones, este bloque consiste inicialmente en 10 réplicas de este *pod* para asegurar el acceso al servicio.

El tercer bloque se genera al inyectar Linkerd a cada componente de la arquitectura generando la capa intermedia de *control plane* el cual se encargará del enrutamiento de todas las peticiones dentro de la arquitectura, las peticiones de los clientes llegaran a este el cual enviara el tráfico a los 10 *pod* del servidor donde el contenedor *Service mesh* actuara como intermediario gestionando el tráfico entrante y saliente del servicio final.

La definición grafica de la arquitectura descrita anteriormente se muestra en la siguiente imagen:

Figura 4. **Arquitectura definida para experimentación**



Fuente: elaboración propia, empleando Draw.io.

3.7. Establecer un entorno de experimentación

Para la definición del entorno de pruebas a realizar se detallan los pasos para su replicación, se puede observar la línea de comandos y una del código de una terminal en la cual se va realizando cada paso, esto se describe a detalle en la sección de apéndices del presente trabajo

3.8. Apache.yaml

Para la definición del despliegue de la arquitectura se establecieron seis bloques que la componen, siendo estas las siguientes:

- Apache *deployment*: el *deployment* es el encargado de mantener al grupo de *pods* del servidor ejecutándose.
- *Client deployment*: el *deployment* del cliente se encarga de mantener a los *pods* que simularan las peticiones de los clientes o sus instancias en ejecución.
- Apache *service*: el servicio de Apache es el encargado de habilitar el acceso a la red para todo el grupo de *pods* en ejecución.
- *Error injector Configmap*: el *configureMap* del inyector de errores se utilizó para generar un *backend* orientado a fallos por medio de Linkerd, devolviendo un error de servidor cuando se inyecte la falla posteriormente.
- *Error injector deployment*: el *deployment* para el inyector de errores se encarga de mantener los *pod* con fallas que se inyecten en ejecución.
- *Error injector service*: el *service* del inyector de errores se encarga de que todos los *pod* con fallas en ejecución tengan acceso a la red particionada del tráfico entrante.

Cada bloque posee distintas etiquetas para la definición de sus objetos, red, funcionamiento entre otros. A continuación, se detallan los más importantes que se observan a lo largo de la plantilla.

- *apiVersion*: es la versión de la API de la que se realizara el despliegue.
- *Kind*: define el tipo de objeto que se crea ya sea de despliegue (*Deployment*), servicio (*Service*) o un mapa de configuraciones (*configMap*).
- *Name*: define el nombre que tendrá el objeto al crearse, cada objeto posee su nombre propio para el manejo del DNS interno de Kubernetes.
- *Namespace*: esta etiqueta sirve para definir un alcance para los elementos en ejecución además de ser una forma de subdividir el clúster para una implementación ordenada y optima.
- *Replicas*: este objeto define la cantidad de imágenes u objetos que se crearan dentro de la arquitectura.
- *Image*: define la imagen base dentro de Docker Hub sobre la cual se despliegan los contenedores dentro de cada *pod*. Esta se define como usuario/imagen según se observó en los pasos previos.

Realizados los pasos descritos en los apéndices, es momento de realizar el despliegue de la arquitectura de prueba previo a la ejecución de los diferentes experimentos, para esto hay que acceder a la carpeta donde está el archivo Apache.yaml y ejecutar el siguiente comando:

```
~$      kubectl apply -f Apache.yaml
```

Figura 5. **Despliegue de arquitectura**

Fuente: elaboración propia, empleando captura de pantalla.

Esto crea cada uno de los componentes descritos anteriormente para observar su estado y funcionamiento dentro del *dashboard* de Linkerd.

Figura 6. **Despliegue exitoso**

Fuente: elaboración propia, empleando captura de pantalla.

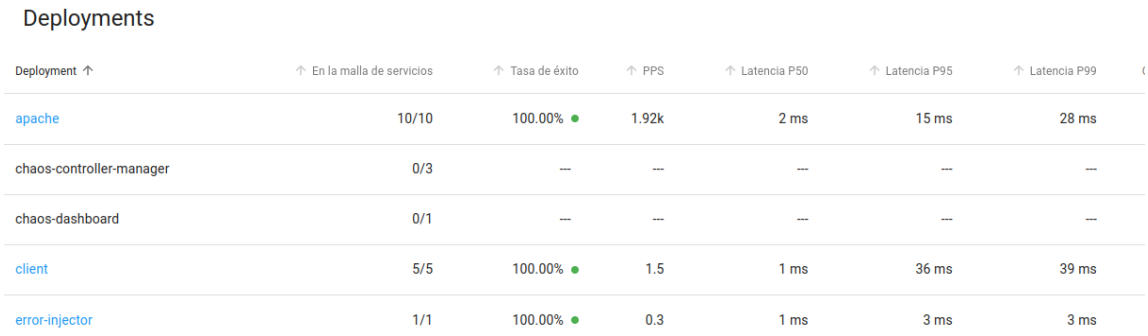
3.9. Definir un estado saludable

Una vez realizado el despliegue dentro del monitor de Linkerd podemos observar el funcionamiento en un estado “ideal” del sistema, se toman algunas métricas como base para la definición de lo que se considera el estado saludable del sistema tomando como base el servidor Apache. Para la toma de datos de un estado saludable se tomaron las métricas obtenidas con la observabilidad de

Linkerd sobre el funcionamiento del sistema con una actualización cada 5 segundos.

A continuación, se muestran las 2 imágenes representativas de los datos mostrados en base a los cuales se define cada métrica.

Figura 7. **Datos de funcionamiento de la arquitectura parte 1**



Deployments

Deployment	En la malla de servicios	Tasa de éxito	PPS	Latencia P50	Latencia P95	Latencia P99
apache	10/10	100.00%	1.92k	2 ms	15 ms	28 ms
chaos-controller-manager	0/3	---	---	---	---	---
chaos-dashboard	0/1	---	---	---	---	---
client	5/5	100.00%	1.5	1 ms	36 ms	39 ms
error-injector	1/1	100.00%	0.3	1 ms	3 ms	3 ms

Fuente: elaboración propia, empleando captura de pantalla.

Figura 8. **Datos de funcionamiento de la arquitectura parte 2**

Fuente: elaboración propia, empleando captura de pantalla.

3.9.1. *Pods* activos

En base a la figura 7 se observó que durante un funcionamiento estable las 10 réplicas del servidor se encuentran activas y en servicio a lo largo del tiempo.

3.9.2. Tasa de éxito

Los servidores responden con una tasa constante de éxito del 100 % a todas las solicitudes que ingresan de parte de los clientes.

3.9.3. Peticiones por segundo

La tasa promedio de solicitudes recibidas en base a la figura 7 es de 1 936 solicitudes por segundo recibidas dentro de los servidores.

3.9.4. Latencia del percentil 50

La latencia máxima en milisegundos para el 50 % más rápido de todas las solicitudes es de 2 ms, es decir que se procesan el 50 % de las solicitudes en menos de 2 ms en un estado normal.

3.9.5. Latencia del percentil 95

La latencia máxima en milisegundos para el 95 % más rápido de todas las solicitudes es de 2 ms, es decir que se procesan el 50 % de las solicitudes en menos de 2 ms en un estado normal.

3.9.6. Latencia del percentil 99

La latencia máxima en milisegundos para el 99 % más rápido de todas las solicitudes es de 28 ms, es decir que se procesan el 99 % de las solicitudes en menos de 28 ms en un estado normal.

3.10. Definir el tipo de experimento que se quiere realizar

Para el inicio de la sección posterior se debe tener conocimiento y las bases de la arquitectura que actualmente se posee y hacerse la pregunta ¿sobre qué aspecto de la arquitectura se realizaran las pruebas?, esta pregunta será el núcleo de los experimentos que se realicen, en base a la teoría en capítulos previos se definieron los puntos dentro de un sistema sobre los cuales se pueden aplicar experimentos de *chaos engineering*.

Para los fines de la ejemplificación de esta guía se realizaron experimentos de trafico de red sobre la arquitectura, fallas en los *pods*, sobre el Kernel de los *pod*, en el tiempo del sistema de los *pods*, estresando el procesador de cada *pod*, sobre peticiones http realizadas y sobre los clústeres de Google Cloud.

4. IMPLEMENTACIÓN DE CAOS

Teniendo una herramienta seleccionada para llevar a cabo los experimentos, una arquitectura definida de la cual se conoce la forma en la que se comporta en un estado natural y las métricas de interés sobre este comportamiento, es tiempo de proceder con la ejecución de distintos experimentos sobre los componentes de un sistema.

4.1. Experimentar

En la fase de experimentación se implementan los pasos o fases recomendadas para la ejecución de experimentos manejando la parte de la ejecución del experimento y el análisis de este en base a las métricas definidas anteriormente.

En los pasos donde sea necesario se realizará la definición que aplique para cada experimento dentro de una viñeta

4.2. Hipótesis

Para esta sección se definió una hipótesis aplicable para cada uno de los experimentos que se ejecutaron previo a la experimentación.

- *faulty-traffic*: con el primer experimento se espera que falle la tasa de respuestas exitosas que genera la arquitectura hacia los clientes, para estabilizar esta tasa de respuestas exitosas/fallidas pero persistentes durante el experimento.

- *Pod-failure*: al desconocer el tipo de falla que se generara dentro de cada *pod* se espera que la tasa de éxitos no se vea afectada en un rango mayor del 10 % en la tasa de respuestas exitosas, que la latencia de respuesta se mantenga constante en los percentiles 50 y 99 y que la cantidad de *pods* activos se mantenga a su totalidad.

- *Pod-kill*: al eliminar un *pod* que se encuentra en funcionamiento se espera que la tasa de respuesta exitosa varia levemente al igual que la latencia de la arquitectura varíe constantemente con cada *pod* que se elimina y recupera sin afectar la latencia del sistema.

- *Network loss*: se espera que este experimento afecte la tasa de éxitos en la arquitectura generado por la pérdida de paquetes, disminuyendo la cantidad de solicitudes por segundo que ingresan.

- *Network delay*: se espera que este experimento provoque un aumento en el *delay* de los percentiles 50 y 99 sin afectar la tasa de éxitos, pero si disminuyendo la cantidad de solicitudes por segundo que ingresan al sistema.

- Unión: con este experimento al mezclar las fallas anteriores, se espera que genere una falla que lleve la cantidad de solicitudes por segundo por debajo de 1, y el *delay* de los percentiles 50 y 99 aumente en más de 100ms por respuesta.

- *Stress*: se espera que este experimento provoque un incremento en el *delay* de los 3 percentiles y una leve disminución en la cantidad de peticiones por segundo sin afectar su tasa de éxito.

- HTTP *faults*: con este experimento se espera la disminución de la tasa de éxitos y de peticiones por segundo.

- *Time faults*: con este experimento se espera que la tasa de peticiones por segundo disminuya y que aumente la latencia en los 3 percentiles, así como una leve disminución en la tasa de éxitos.

- *Kernel faults*: con este experimento se espera que la tasa de peticiones por segundo disminuya considerablemente y que aumente la latencia en los 3 percentiles, de la misma forma, temporalmente se verá afectada en gran medida la tasa de éxitos.

4.2.1. Elegir un componente y comprenderlo

A continuación, se detalla el funcionamiento general de cada componente para comprender su funcionamiento y rol dentro de la arquitectura.

- Red: la red al ser la encargada del enrutamiento de todas las peticiones entrantes, la comunicación entre los servicios debe funcionar de forma óptima para minimizar tiempos de respuesta y la disponibilidad del sistema. Manteniendo una baja saturación o *delay* en cada una de las respuestas.

- *Pod*: el *pod* es la unidad más pequeña dentro de Kubernetes, el cual consta de un contenedor del servidor Apache y un contenedor *sidecar* para el manejo de las peticiones que llegan al *pod* y la respuesta saliente, es decir el por será el elemento que o instancia que se encuentra brindando el servicio hacia los clientes.

- Kernel**:** es el *software* base de ejecución de cualquier sistema operativo que se encarga de mediar entre los procesos y el *hardware* que cada equipo puede tener disponible, es el núcleo de ejecución de cada contenedor que se ejecuta en los *pods*.

4.2.2. Identificar servicios/componentes críticos

A continuación, detallan los componentes/servicios críticos o que se cree se verán afectados con la ejecución de cada experimento.

Tabla XII. **Servicios/componentes críticos**

Experimento	Puntos críticos
Faulty traffic	Red
Pod-failure	*Pod*/red
Pod-kill	*Pod*
Network loss	Red
Network delay	Red
Stress	*Pod*/rendimiento
Http-faults	Peticiones
Time-faults	*Pod*/red
Kernel-faults	*Pod*
Union	Posible daño a toda la arquitectura

Fuente: elaboración propia, empleando Microsoft Word.

4.2.3. Verificar ¿Qué puede salir mal?, ¿sabemos que pasara si falla?

En la mayoría de los experimentos dada la definición de las plantillas individuales es predecible cada resultado en cierta medida, pero al realizar una ejecución con todos los experimentos, se desconoce la magnitud de la falla, hay que considerar que se puede generar una falla general la cual no permita que el sistema se recupere por sí mismo hasta volver a ejecutar toda la arquitectura como la peor de las situaciones.

En el caso de realizar un experimento de muy gran impacto en un ambiente de producción es necesario tener todo listo para ejecutar la arquitectura nuevamente o experimentar un *downtime* inesperado.

4.3. Seleccionar un experimento

Se debe tener claro con que aspecto del sistema se desea experimentar, como ejemplo sobre la arquitectura definida se estará experimentando con la red y los *pod* del sistema. Para los fines de experimentación del presente trabajo se realizaron 6 experimentos utilizando las herramientas descritas anteriormente.

- Inyección de *faulty-traffic*: este experimento consiste en la inyección de un *traffic splitter* con Linkerd únicamente en donde se genera un *pod* adicional que retorna una respuesta fallida afectando el tráfico y las métricas de funcionamiento del sistema.

- *Pod-failure*: este experimento genera fallas aleatorias dentro de un *pod*, especificando un rango de tiempo concurrente en el que se desea que se repita la falla y un tiempo de duración para cada falla.

- *Pod-kill*: este experimento destruye un *pod* en ejecución indicando un rango de tiempo en el cual se desea que se repita la incidencia.

- *Network loss*: este experimento consiste en la perdida de paquetes de forma aleatoria.

- *Network delay*: este experimento consiste en generar una acción que retrasara el envío de mensajes dentro de la red de los *pods*.

- *Stress*: este experimento consiste en la simulación de estrés, es decir carga adicional sobre determinados contenedores de forma continua, así como lecturas y escrituras simuladas para incrementar la carga en la memoria.

- Http *faults*: este experimento simula escenarios fallidos durante las *transacciones* HTTP y sus respuestas.

- *Time faults*: este tipo de experimento simula un desfase de horario en determinados *pods* provocando una pérdida de sincronía entre los componentes.

- Kernel *faults*: este experimento simula inyecta fallas de forma simulada en el Kernel de Linux afectando a uno o muchos *pods*. Cabe destacar que este experimento es arriesgado dado que los demás *pods* comparten el mismo Kernel por lo que este experimento no debe realizarse en un ambiente de producción.

- Union: denominado de esta forma porque combina los efectos del *faulty-traffic, pod-failure* y pod-kill en un solo experimento para aumentar el rango de daño en el sistema.

4.4. Determinar un radio de impacto pequeño

Para la ejecución de los experimentos es necesario establecer los límites que tendrá cada uno, esto quiere decir el nivel de daño que se desea causar con cada experimento, iniciando desde un punto pequeño para ir aumentando el grado de daño en cada repetición del experimento. Los radios de impacto iniciales para cada experimento se detallan a continuación.

- *faulty-traffic*: se definió inicialmente un balance del 90 % de peticiones hacia los *pods* para una ejecución correcta y el 10 % hacia un *pod* con *backend* fallido.

- *Pod-failure*: se definió inicialmente que la falla se produzca cada 10 segundos y la falla generada dure 15 segundos.

- *Pod-kill*: se definió que la falla se genere cada 3 segundos.

- *Network loss*: se definió una pérdida de 30 paquetes, esta falla durará 30 segundos y se repetirá continuamente cada 50 segundos.

- *Network delay*: se definió una latencia agregada de 100 ms, un *jitter* de 10ms con duración de la falla de 15 segundos. Esta falla se repetirá de forma continua cada 30 segundos.

- *Stress*: se definió una carga sobre la memoria de los *pod* de 256 MB de forma persistente hasta que el experimento se detenga de forma manual luego de 3 minutos.

- Http *faults*: se definió una generación de peticiones abortadas durante un tiempo de 3 minutos sobre las peticiones *POST* a través del puerto 80 de los servidores *web*.

- *Time faults*: se definió un cambio en el tiempo que manejan los *pod* por 10 minutos y 100 nano segundos, este experimento se ejecutara de forma continua por 5 minutos hasta que el experimento se detenga de forma manual.

- Kernel *faults*: se definió la inyección de una falla sobre la función __x64_sys_mount el cual genera una filtración de memoria en el sistema, este experimento se detendrá manualmente luego de un minuto de ejecución.

- Unión: siendo este experimento una combinación de las 5 fallas mencionadas anteriormente se aumentó su rango de impacto, generando un tráfico fallido del 20 %, una falla en *pods* con duración de 15 segundos que se ejecutara cada 10 segundos. Una eliminación de *pods* cada 3 segundos. Una pérdida de 30 paquetes como duración de 30 segundos para que se ejecute cada 50 segundos. Y por último un *delay* en la red con latencia de 100ms, un *jitter* de 10ms con duración de 15 segundos para esta falla y una ejecución constante cada 30 segundos.

4.5. Determinar métricas para evaluar

Las métricas para evaluar durante los experimentos son las que definieron el estado saludable de la arquitectura para los fines de experimentación, los cuales se muestran a continuación:

- *Pods* activos
- Tasa de éxito del servicio
- Peticiones por segundo
- Latencia del percentil 50
- Latencia del percentil 95
- Latencia del percentil 99

4.6. Ejecutar el experimento (crear caos)

Para la ejecución de cada experimento se accede a la carpeta de Experimentos y se ejecuta el siguiente comando:

```
~$ kubectl apply -f <Nombre.yaml>
```

Y para detener cada experimento es necesario ejecutar:

```
~$ kubectl delete -f <Nombre.yaml>
```

4.7. Verificar resultados

Realizado cada experimento es necesario verificar el daño esperado contra los resultados que realmente se obtuvieron en base al radio de impacto

definido, a continuación, se detallan los resultados obtenidos con cada experimento y como estos afectaron las métricas del sistema.

4.7.1. faulty-traffic

Para este experimento se inyecto un divisor de tráfico en el cual se definió en el archivo la división de las cargas de envió de información a los 2 servicios (Apache y *error-injector*).

Figura 9. **Archivo de experimento 1**

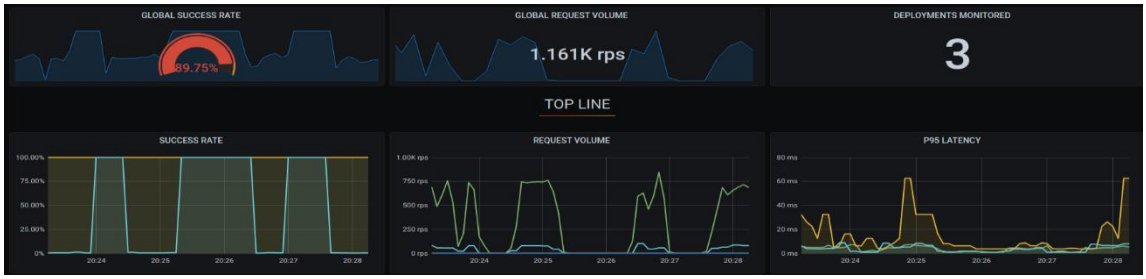
```yaml
apiVersion: split.smi-spec.io/v1alpha1
kind: TrafficSplit
metadata:
  name: error-split
  namespace: chaos-testing
spec:
  service: apache
  backends:
  - service: apache
    weight: 900m
  - service: error-injector
    weight: 100m
```

Fuente: elaboración propia, empleando captura de pantalla.

Al iniciar la ejecución del experimento la tasa de éxito se empezó a disminuir gradualmente hasta comenzar a estabilizarse y mantenerse en una cifra cercana al 90 % de peticiones exitosas, la cantidad de peticiones por segundo se mantuvo variable entre el estado saludable y disminuyendo temporalmente por debajo de las 500 peticiones por

segundo. Ambas latencias tanto para el percentil 50 y 99 disminuyeron por debajo de la mitad de su estado saludable.

Figura 10. **métricas experimento 1**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.2. Pod-failure

Para este experimento se definió una duración de 15 segundos de impacto y que se repitiera el experimento cada 10 segundos aplicado hacia la aplicación de Apache.

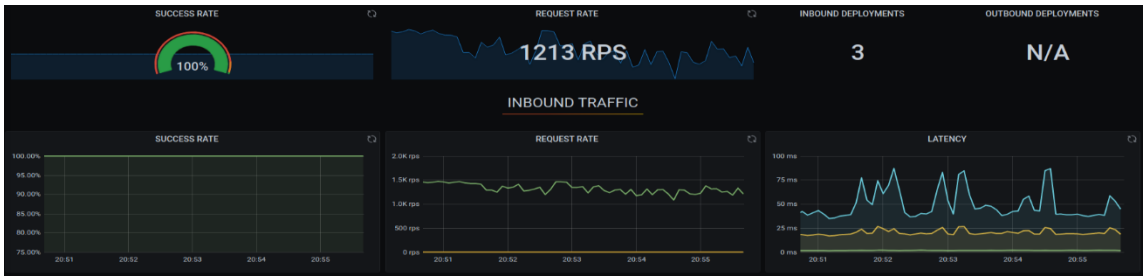
Figura 11. **Archivo de experimento 2**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: pod-failure-example
  namespace: chaos-testing
spec:
  action: pod-failure
  mode: one
  value: ''
  duration: '15s'
  selector:
    labelSelectors:
      'app': 'apache'
  scheduler:
    cron: '@every 10s'
```

Fuente: elaboración propia, empleando captura de pantalla.

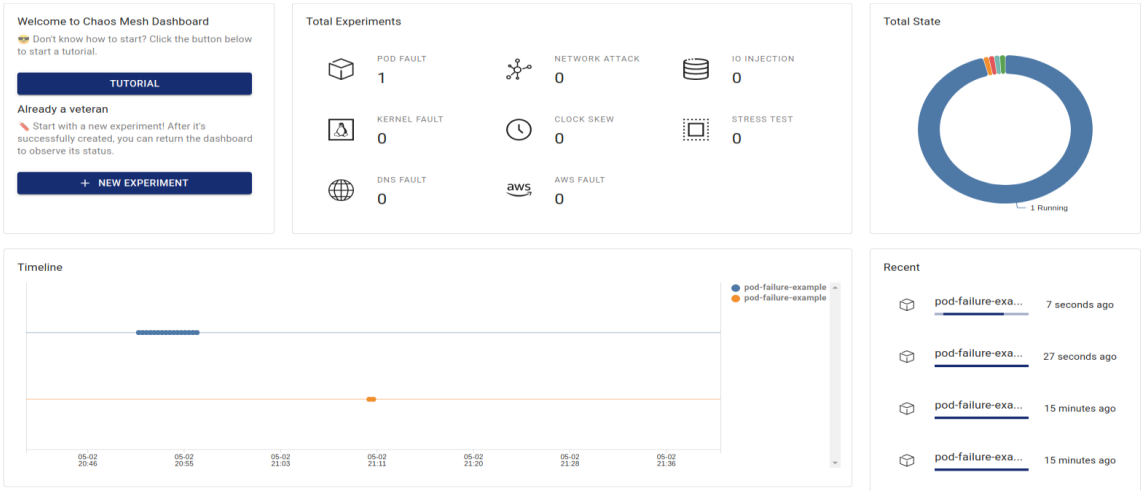
La tasa de éxitos no se vio afectada debido a que los *pod* que continuaban en funcionamiento se encargaron de mantener el correcto estatus del sistema. La tasa de peticiones por segundo se mantuvo cercana a la media definida en el estado saludable del sistema. El *delay* perteneciente el percentil 50 se mantuvo estable pero la del percentil 99 aumentaba temporalmente cada vez que el experimento generaba una falla en un nuevo *pod*, al terminar la ejecución este *delay* se estabilizaba.

Figura 12. **Métricas Experimento 2**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 13. **Estado del experimento 2**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.3. Pod-kill

Para este experimento se definió que se matara un *pod* cada 3 segundos perteneciente a la aplicación de Apache.

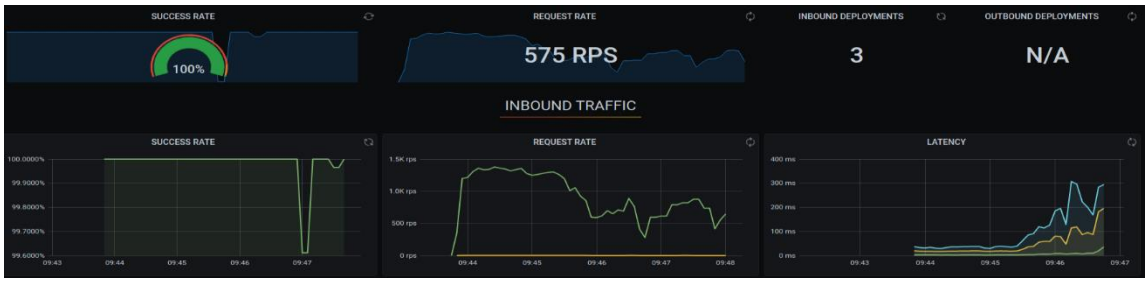
Figura 14. **Archivo de experimento 3**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: pod-kill-demo
  namespace: chaos-testing
spec:
  action: pod-kill
  mode: one
  selector:
    namespaces:
      - chaos-testing
    labelSelectors:
      'app': 'apache'
  scheduler:
    cron: '@every 3s'
```

Fuente: elaboración propia, empleando captura de pantalla.

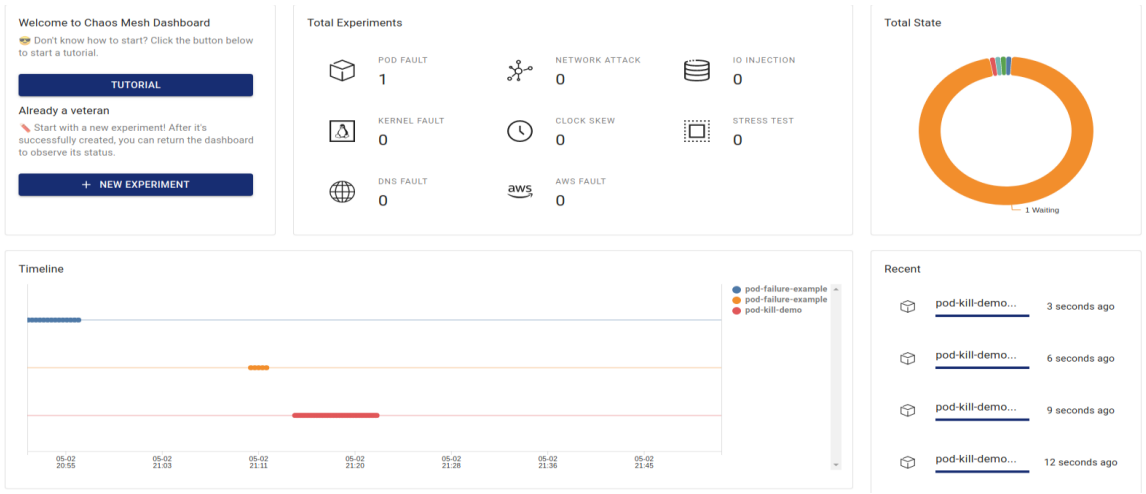
La cantidad de *pods* activos se mantuvo con una reactivación casi instantánea, pero la latencia de la arquitectura se vio afectada en cada eliminación y creación de un nuevo *pod*, superando la latencia del percentil 99 en más de 300 ms en su pico más alto. La tasa de éxitos disminuía levemente con cada *pod* eliminado y la cantidad de peticiones por segundo disminuyo por debajo de 600.

Figura 15. **Métricas experimento 3**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 16. **Estado experimento 3**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.4. Network loss

Para este experimento se definió una pérdida de 30 paquetes con duración del experimento de 30 segundos para que se repitiera esta secuencia cada 50 segundos, afectando la comunicación del servicio de Apache, en el cual

tenía que perder 30 paquetes de forma aleatoria afectando la entrega de paquetes dentro del sistema.

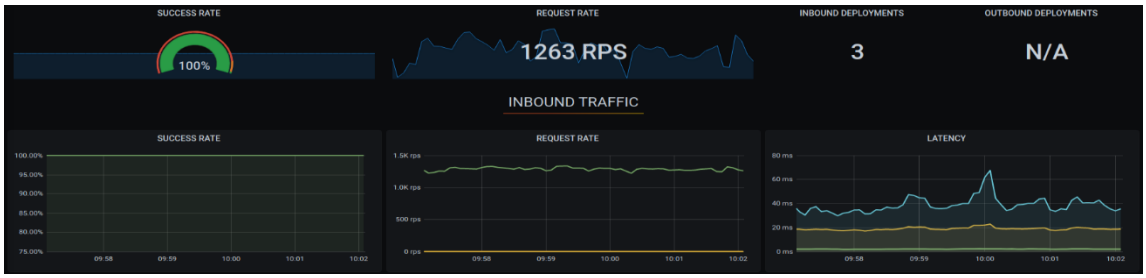
Figura 17. **Archivo de experimento 4**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: network-loss-example
  namespace: chaos-testing
spec:
  action: loss
  mode: one
  selector:
    namespaces:
      - chaos-testing
    labelSelectors:
      'app': 'apache'
  loss:
    loss: "30"
    correlation: "30"
  duration: "30s"
  scheduler:
    cron: "@every 50s"
```

Fuente: elaboración propia, empleando captura de pantalla.

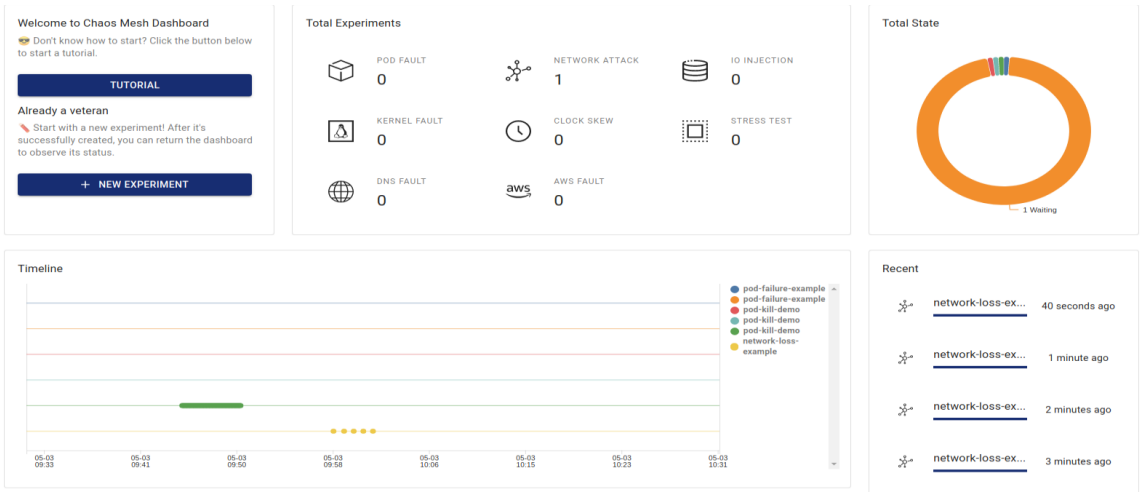
La tasa de éxitos no se vio afectada por el experimento, las peticiones por segundo disminuyeron levemente sin alejarse de la media de su estado saludable. La latencia del percentil 50 no se vio afectada mientras que la del percentil 99 presentaba picos en aumento durante el tiempo de la perdida de paquetes únicamente sin afectar de forma considerable al sistema.

Figura 18. **Métricas experimento 4**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 19. **Estado experimento 4**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.5. Network delay

Para este experimento se definió una latencia de 100 ms, un *jitter* de 10ms, para que el experimento durara 15 segundos y esta secuencia se repitiera

cada 30 segundos, afectando la comunicación del servicio de Apache en el cual debía generar retraso en la entrega de paquetes dentro del sistema.

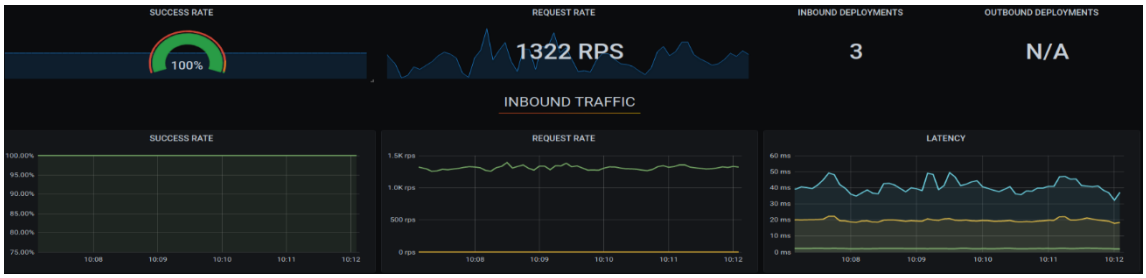
Figura 20. **Archivo de experimento 5**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: network-delay-example
  namespace: chaos-testing
spec:
  action: delay
  mode: one
  selector:
    namespaces:
      - chaos-testing
    labelSelectors:
      'app': 'apache'
  delay:
    latency: "100ms"
    correlation: "25"
    jitter: "10ms"
  duration: "15s"
  scheduler:
    cron: "@every 30s"
```

Fuente: elaboración propia, empleando captura de pantalla.

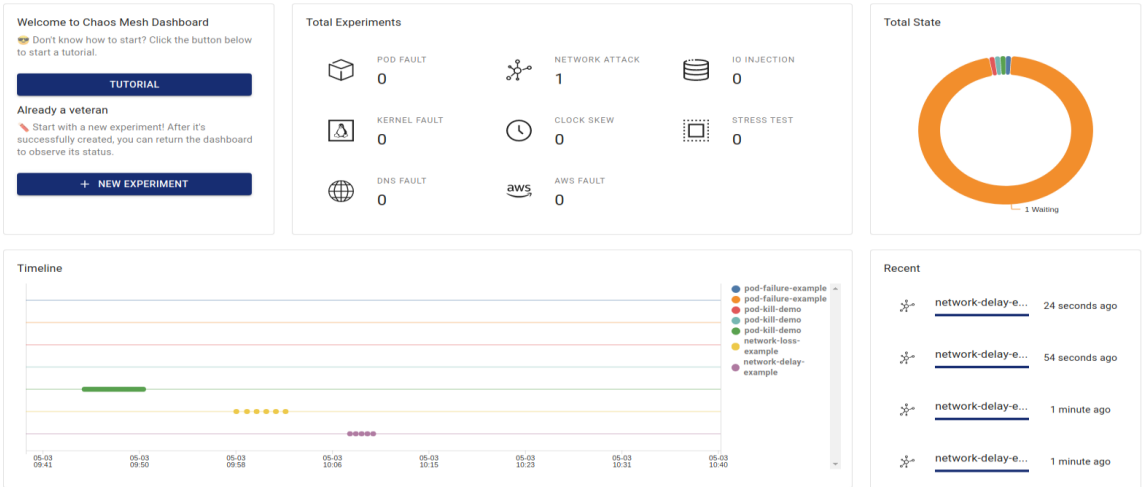
El sistema no se vio afectado de forma considerable, soporto el experimento sin alterar su estado saludable.

Figura 21. **Métricas experimento 5**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 22. **Estado experimento 5**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.6. Stress

Para este experimento se agregó una carga de 256 MB sobre la memoria de los *pods* en ejecución. El experimento duró 3 minutos y debía disminuir la cantidad de peticiones por segundo.

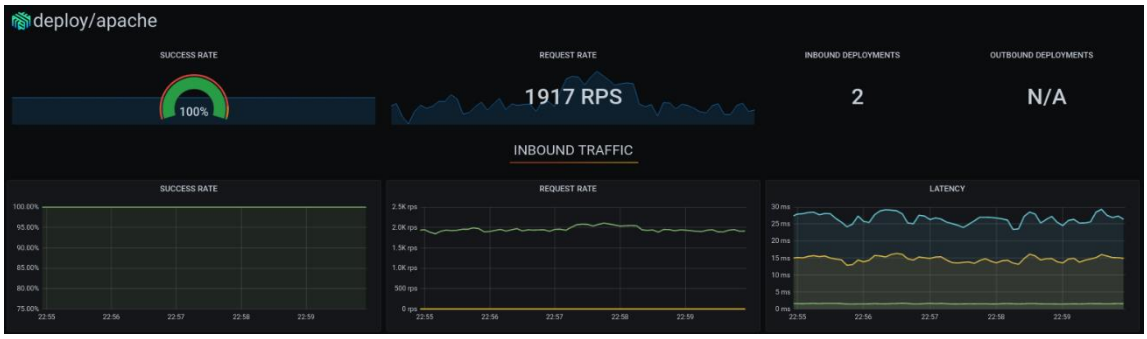
Figura 23. **Archivo de experimento 6**

```
apiVersion: chaos-mesh.org/v1alpha1
kind: StressChaos
metadata:
  name: memory-stress-example
  namespace: chaos-testing
spec:
  mode: one
  selector:
    labelSelectors:
      'app': 'apache'
  stressors:
    memory:
      workers: 4
      size: '256MB'
```

Fuente: elaboración propia, empleando captura de pantalla.

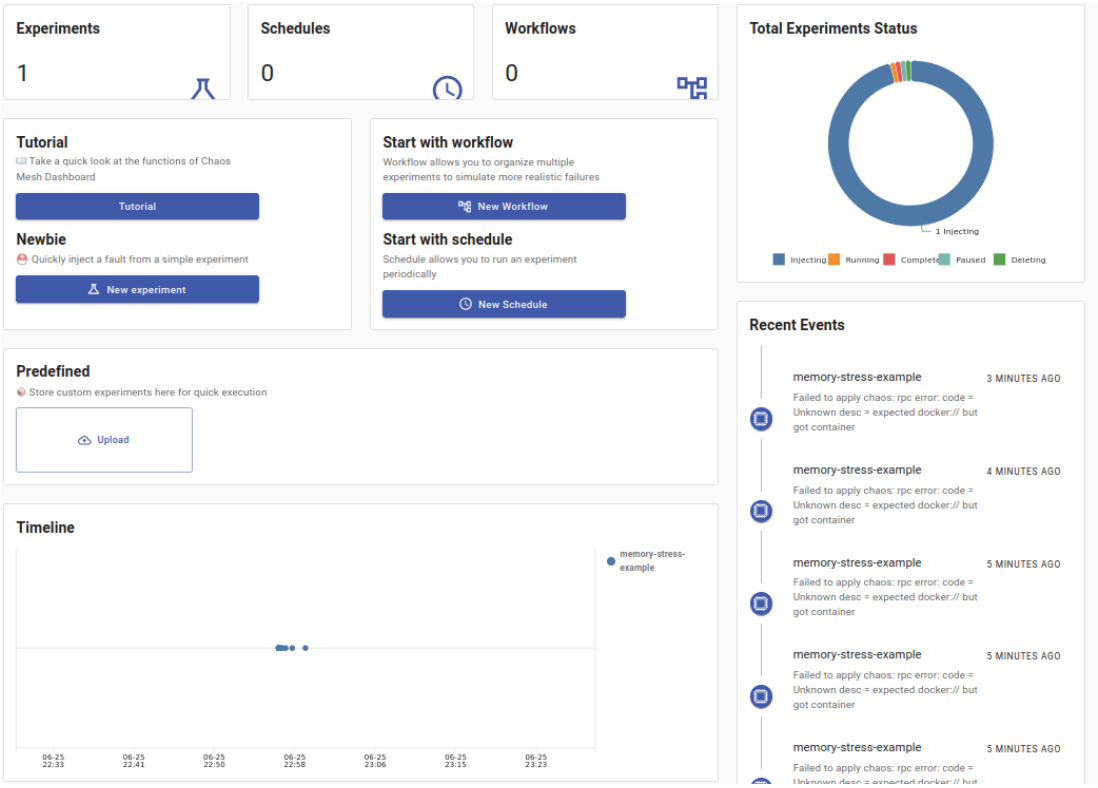
El funcionamiento no se vio afectado, en mayor medida la cantidad de peticiones por segundo y la tasa de éxito se mantuvo estable, los *pods* no se vieron afectados por la carga adicional en su memoria.

Figura 24. **Métricas experimento 6**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 25. **Estado experimento 6**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.7. Http faults

Para este experimento se definió una falla constante aleatoria de una petición abortada sobre el servidor con una duración de 5 minutos para la falla de forma constante, afectando la tasa de peticiones atendidas y la tasa de éxito.

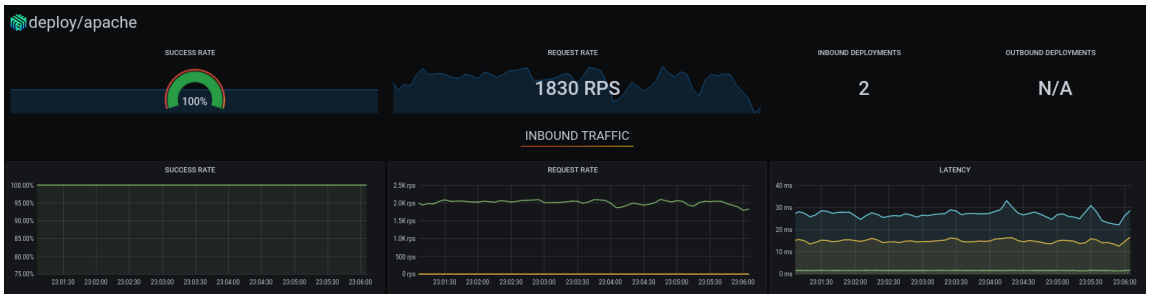
Figura 26. **Archivo de experimento 7**

```
apiVersion: chaos-mesh.org/v1alpha1
kind: HTTPChaos
metadata:
  name: test-http-chaos
spec:
  mode: all
  selector:
    labelSelectors:
      'app': 'apache'
  target: Request
  port: 80
  method: GET
  path: /api
  abort: true
  duration: 5m
```

Fuente: elaboración propia, empleando captura de pantalla.

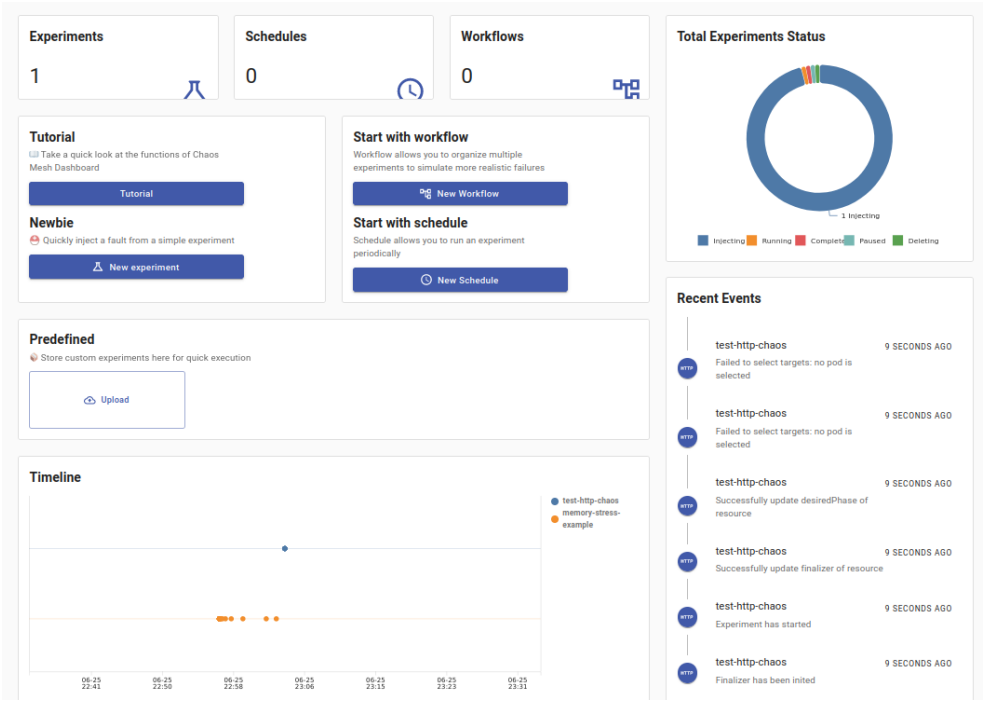
El experimento afecto levemente la tasa de peticiones por segundo atendidas disminuyendo en promedio 100 peticiones por segundo menos que su estado estable sin afecta la tasa de éxito sobre las peticiones atendidas.

Figura 27. **Métricas experimento 7**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 28. **Estado experimento 7**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.8. Time faults

Para este experimento se definió la inyección de un desfase en el horario de los *pods* de 10 minutos y 100 nano segundos para perder la sincronía entre sí, con una ejecución de 5 minutos constantes con la falla, en busca de afectar el recibimiento de las peticiones por segundo y la latencia en las respuestas de parte del servicio.

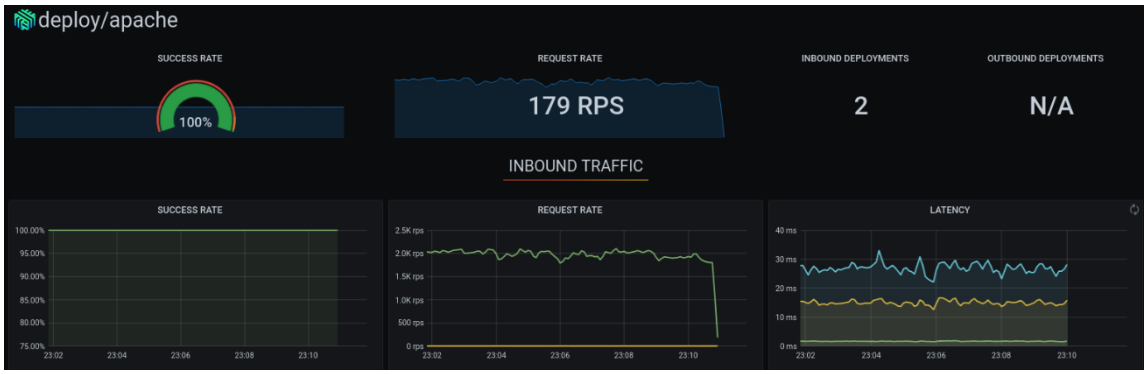
Figura 29. **Archivo de experimento 8**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: TimeChaos
metadata:
  name: time-shift-example
  namespace: chaos-testing
spec:
  mode: one
  selector:
    labelSelectors:
      'app': 'apache'
  timeOffset: '-10m100ns'
```

Fuente: elaboración propia, empleando captura de pantalla.

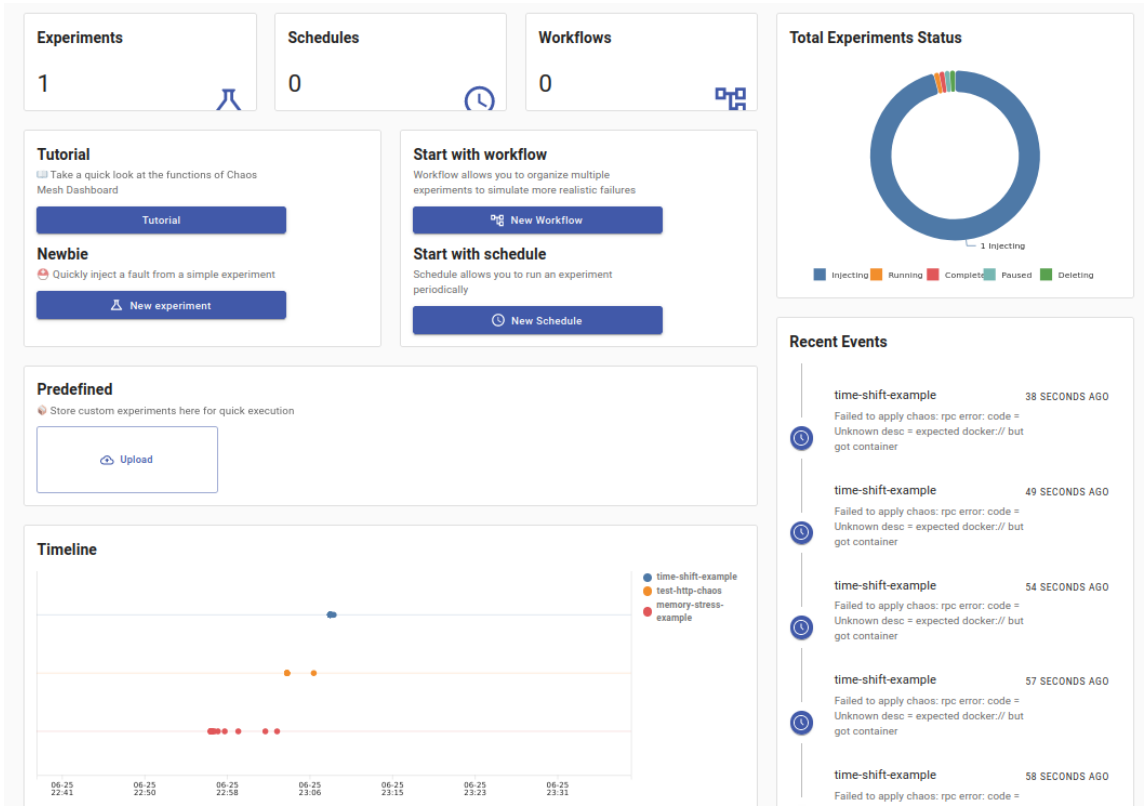
El experimento presento un fuerte impacto en la tasa de peticiones por segundo, dado que al perder la sincronía de horario muchos *pods* activos eran incapaces de responder a las peticiones disminuyendo drásticamente esta cantidad sin afectar la tasa de éxito en los *pods* que respondían correctamente y su latencia para los 3 percentiles ante la situación.

Figura 30. **Métricas experimento 8**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 31. **Estado experimento 8**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.9. Kernel *faults*

Para este experimento se inyecto una falla sobre la función __x64_sys_mount la cual provoca una filtración sobre la memoria del sistema generando fallas en diversas partes del sistema, dicha falla persistió por 5 minutos donde se esperaba un incremento en la latencia de los 3 percentiles y una disminución den la tasa de peticiones por segundo.

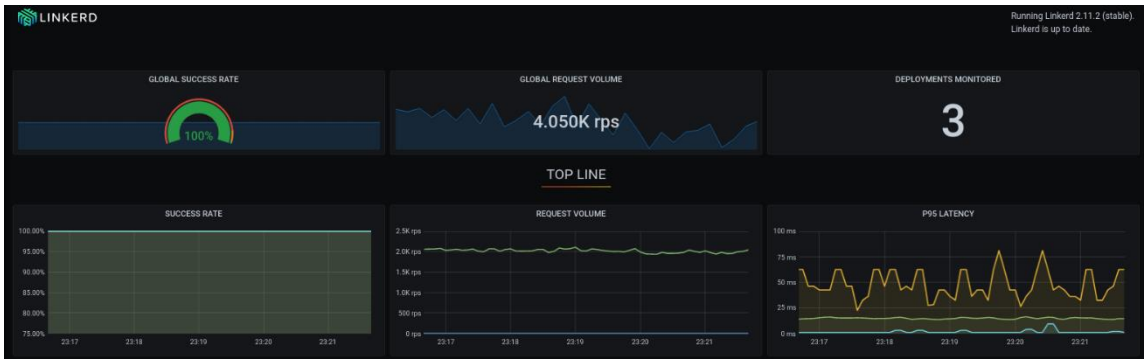
Figura 32. **Archivo de experimento 9**

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: KernelChaos
metadata:
  name: kernel-chaos-example
  namespace: chaos-testing
spec:
  mode: one
  selector:
    namespaces:
      - chaos-testing
  failKernRequest:
    callchain:
      - funcname: '__x64_sys_mount'
    failtype: 0
```

Fuente: elaboración propia, empleando captura de pantalla.

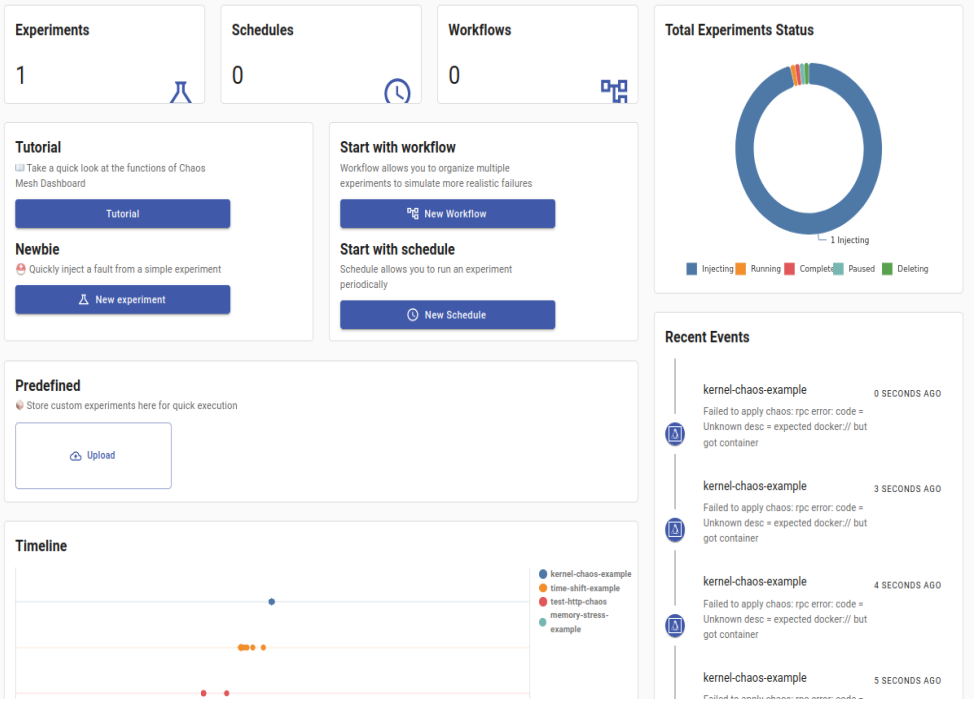
La latencia del percentil 50 no se vio afectada, sin embargo, la latencia de los percentiles 95 y 99 incremento por encima de los 50 ms en ambos y debido al daño que tenía el Kernel de los *pods* la tasa de peticiones por segundo presento una variación en los datos de entre 2 000 a 4 000 peticiones por segundo.

Figura 33. **Métricas experimento 9**



Fuente: elaboración propia, empleando captura de pantalla.

Figura 34. **Estado experimento 9**



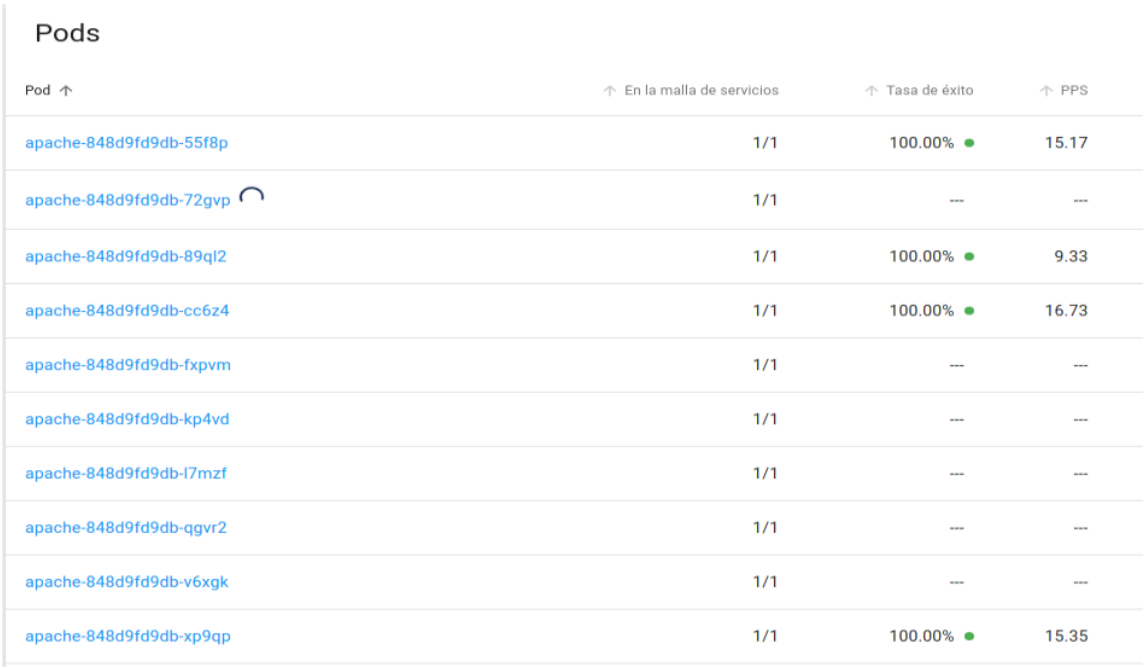
Fuente: elaboración propia, empleando captura de pantalla.

4.7.10. Unión

Este experimento fue una combinación cinco de los experimentos previos, uniendo los archivos Yaml descritos en cada uno de los experimentos anteriores.

En 10 segundos se notó la caída de las peticiones por segundo por debajo de las 200, luego de 1 minuto hubieron caídas por debajo de 100 peticiones por segundo. La cantidad de *pods* funcionales comenzaba a fallar, llevando a no recibir peticiones en más de la mitad de los *pods* durante las fallas.

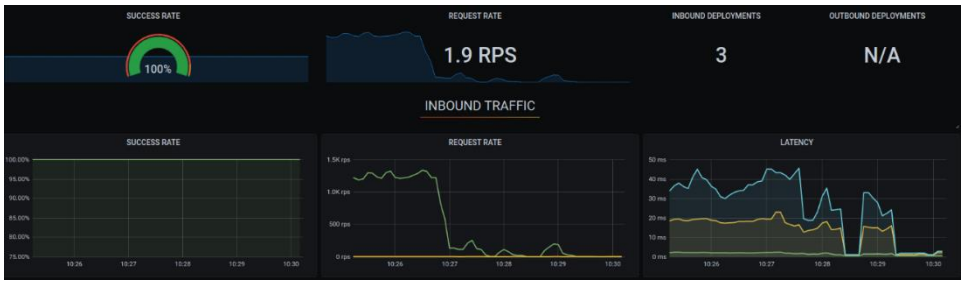
Figura 35. **Estado de *pods* experimento 10**



Fuente: elaboración propia, empleando captura de pantalla.

La latencia de ambos percentiles cayó a 0 en un punto donde todos los *pods* presentaban fallas y llevan a aceptar únicamente 2,1 peticiones por segundo o menos. la tasa de éxito no vario dado que por el balanceador de la carga inherente de Kubernetes se redirigió el tráfico únicamente a los *pods* que quedaban en funcionamiento. El sistema presento una falla de rendimiento total que llevo a sus métricas por los suelos.

Figura 36. **Métricas experimento 10**



Fuente: elaboración propia, empleando captura de pantalla.

4.7.11. Resumen de experimentos

A continuación, se muestra una tabla con los resultados obtenidos sobre las meticas en comparación a su estado saludable ubicado en la parte baja de la tabla, en dicha tabla se puede observar la comparativa en cómo se vio afectado el sistema con la ejecución de cada experimento realizado.

Tabla XIII. **Resultado de experimento realizados**

Experimento	Tasa de éxito	PPS	Latencia P50	Latencia P95
Faulty-traffic	90 %	500	< 5 ms	< 60 ms
Pod-failure	100 %	1 213	< 2 ms	< 75 ms
Pod-kill	100 % - 0 %	> 600	< 7 ms	< 300 ms
Network loss	100 %	1 263	2 ms	< 60 ms
Network delay	100 %	1 322	2 ms	< 37 ms
Stress	100 %	1 917	2 ms	29 ms
Http faults	100 %	1 830	2 ms	29 ms
Time faults	100 %	179	2 ms	29 ms
Kernel faults	100 %	< 2 000	<50	<50
Union	100 %	> 1.9	0 ms	0 ms
Estado saludable	100 %	1 354	2 ms	32 ms

Fuente: elaboración propia, empleando Microsoft Word.

4.7.12. Tiempo para detectar

Este tiempo será un indicador de desempeño que muestra el tiempo promedio necesario para poder detectar la falla dentro de nuestro sistema o poder identificar los primeros rasgos que nos muestren una alteración dentro del sistema siendo este tiempo desde 0 cuando se empieza a ejecutar el experimento hasta N, donde N será el tiempo transcurrido desde el inicio del experimento hasta el punto donde las métricas del estado saludable comienzan a verse afectadas.

- *faulty-traffic*: monitoreando las métricas dentro del *dashboard* de Linkerd, el tiempo para visualizar la falla fue menor a 10 segundos desde que se alteró el estado saludable del sistema.

- *Pod-failure*: monitoreado las métricas dentro del *dashboard*, tomo alrededor de 1 minuto cuando se empezó a notar un aumento en la latencia y descontrol en los *pods* del servidor.

- *Pod-kill*: aproximadamente 1 minuto donde se empezó a notar los primeros cambios en la latencia de las respuestas, la tasa de éxitos y la cantidad de peticiones por segundo.

- *Network loss*: 3 minutos para ver un leve aumento en la perdida de paquetes.

- *Network delay*: 1 minuto por la leve variación que se presentó en la latencia del percentil 99 sin alejarse del estado saludable.

- *Stress***:** aproximadamente 2 minutos por la leve variación en la cantidad de peticiones por segundo.

- Http *faults***:** aproximadamente 1 minuto por el leve descenso en la cantidad de peticiones por segundo.

- *Time faults***:** se observaron cambios a los 20 segundos de iniciado el experimento mostrando un drástico descenso en la cantidad de peticiones por segundo.

- *Kernel faults***:** a los 10 segundos de iniciado el experimento se observó una variación intermitente en la latencia del percentil 95 y 99, así como un incremento abrupto en la tasa de peticiones registradas por el servicio.

- Unión: 10 segundos para la caída de las peticiones y todas las métricas un minuto después.

4.7.13. Tiempo para auto recuperarse

Este tiempo será el que tome al sistema estabilizar por sí mismo sus métricas o recuperando los *pods*/contenedores caídos, en caso sea posible para él. Este tiempo se toma desde el momento en el que se detecta una falla en el sistema, hasta que este consigue estabilizarse por sí solo del efecto de alguno de los experimentos.

- *faulty-traffic*: el sistema tardaba aproximadamente 30 segundos en estabilizar las peticiones hacia el tráfico funcional, pero esta falla persistía y se continuaba generado por lo que hasta que no se solucionó la falla completamente el sistema no se recuperó totalmente.

- *Pod-failure*: no fue posible la auto recuperación del sistema ya que la falla continuaba periódicamente hasta el fin de la falla.

- *Pod-kill*: inmediato, cada *pod* que el experimento eliminaba, el mismo entorno lo levantaba nuevamente de forma automática para recuperarse.

- *Network loss*: no fue posible dado que la perdida de paquetes continuaba generándose, sin afectar en mayor medida el sistema.

- *Network delay*: el sistema no se vio afectado, pero la falla persistió hasta finalizar el experimento.

- *Stress*: el sistema no se vio afectado en mayor medida y siguió operando de forma correcta durante la persistencia de la falla.

- Http *faults*: el sistema no se vio afectado en mayor medida y siguió operando de forma correcta durante la persistencia de la falla.

- *Time faults*: debido a la perdida de sincronía del servicio no fue posible para el recuperarse por sí mismo.

- Kernel *faults*: al ser un daño al Kernel de ciertos *pod* el servicio intentaba periódicamente estabilizar el funcionamiento activando y desactivando los *pods* dañados para continuar con el funcionamiento, este comportamiento persistió hasta el final del experimento.

- Unión: no fue posible.

4.7.14. Tiempo de recuperación

Este tiempo será el que le tome al sistema estabilizarse completamente luego de terminado algún experimento o falla que lo haya afectado. Se toma como su punto de conteo desde el momento en el que se termina de ejecutar un experimento hasta el punto en el cual regresa a su estado saludable.

- *faulty-traffic*:
    - Parcial: de forma constante cada 30 segundos se estabilizaban las métricas, pero la falla se repetía.
    - Total: luego de ser resuelta la falla el sistema tardo 5 segundos en estabilizar el porcentaje de sus peticiones exitosas.

- *Pod-failure*:
    - Parcial: de forma constante al terminar la falla en un *pod*.
    - Total: luego de 10 segundos de haber finalizado la ejecución del experimento el sistema se estabilizaba.

- *Pod-kill*:
    - Parcial: inmediato pero la falla se seguía produciendo
    - Total: 10 segundos posterior a la finalización del experimento

- *Network loss*:
    - Parcial: al finalizar el tiempo de la perdida de paquetes
    - Total: inmediato al finalizar el experimento

- *Network delay*:
    - Parcial: no fue posible
    - Total: inmediato al finalizar el experimento

- *Stress*:
    - Parcial: no fue posible, aunque el servicio no se vio afectado de forma considerable.
    - Total: inmediato al finalizar el experimento.

- Http *faults*:
    - Parcial: en el intervalo de la falla
    - Total: inmediato al finalizar el experimento

- *Time Faults*:
    - Parcial: no fue posible
    - Total: inmediato al finalizar el experimento

- Kernel *faults*:
    - Parcial: no fue posible
    - Total: inmediato al finalizar el experimento

- Unión:
    - Parcial: no fue posible, presento una falla completa.
    - Total: aproximadamente 10 minutos en lo que el sistema comenzó a elevar nuevamente sus métricas, 12 minutos para una estabilización completa del sistema.

4.7.15. Tiempo de falla

Este será el tiempo durante el cual una falla o experimento estuvo afectando al sistema antes de ser resuelto o detenido completamente. Iniciando desde el momento en el cual se detecta que una falla afecta al sistema hasta que la falla se da por concluida.

En todos los experimentos se mantuvo un tiempo de ejecución de entre 3 y 5 minutos, equivalente al tiempo que duro la falla afectando al sistema.

4.7.16. Escalar o presionar

Al verificar los resultados es momento de tomar la decisión si se considera que la arquitectura soporto de forma aceptable dentro de las políticas de funcionamiento esperado los experimentos, de ser este el caso se puede repetir cada experimento aumentando el radio de impacto según se considere necesario para verificar nuevamente su reacción ante el cambio. En caso contrario y se considere que la arquitectura no soporto lo suficiente para ser considerada

funcional en el entorno actual, será necesario tomar decisiones para su mejora y repetir el proceso hasta llegar a un punto de aceptación.

En los diez casos mostrados se observa que la arquitectura soporto con éxito 4 de los 10 experimentos, por lo cual lo recomendable es escalar la arquitectura, dado que se comprobó que la arquitectura es propensa a diversas fallas en las condiciones mostradas.

4.8. Mejorar

En base al paso anterior ya sea mejorar la arquitectura o escalarla para que soporte condiciones más duras, es necesario tener en cuenta que siempre se debe buscar una mejora continua en la misma, aunque el sistema soporte los ataques o experimentos se debe llevar al límite este sistema para encontrar sus puntos de quiebre y buscar una mejora para aumentar su tolerancia a fallos, resiliencia y disminuir los tiempos de caída en la mayor medida posible.

La finalidad de *chaos engineering* es buscar mejorar cualquier sistema incluso si este ya se encuentra funcional bajo condiciones normales.

5. POPULARIDAD ACTUAL DE *CHAOS ENGINEERING*

Ciertamente *chaos engineering* es un tema muy joven todavía, tanto a nivel nacional como en el extranjero, la documentación sobre el tema aun es escasa y ambigua por lo que durante los meses de desarrollo de la presente investigación se mantuvo contacto con diferentes organizaciones internacionales para conversar sobre el tema, donde fuimos no más de 40 personas por reunión de varios países. En el ámbito guatemalteco conversando con diferentes profesionales del área y estudiantes no se observaba conocimiento del tema por lo que se realizó la recopilación de información básica orientada a personas que laboran en el área de informática.

5.1. Definición de encuesta realizada

Se realizo un formulario dentro de Google Forms para la recopilación de datos sobre el conocimiento y la popularidad que actualmente posee *chaos engineering* dentro de Guatemala. Para esto se definieron 11 preguntas que buscaban abarcar las generalidades sobre el tema.

Las mismas se definen en la sección de aprendices junto con sus posibles respuestas.

5.2. Resultados

La encuesta fue realizada por un total de 47 personas seleccionadas al azar, que laboran o estudiaran en el ámbito informático en el territorio guatemalteco. Posterior a la pregunta 1 se brinda una breve descripción y finalidad de *chaos engineering* para que los participantes en la encuesta tuvieran una pequeña noción de sus beneficios.

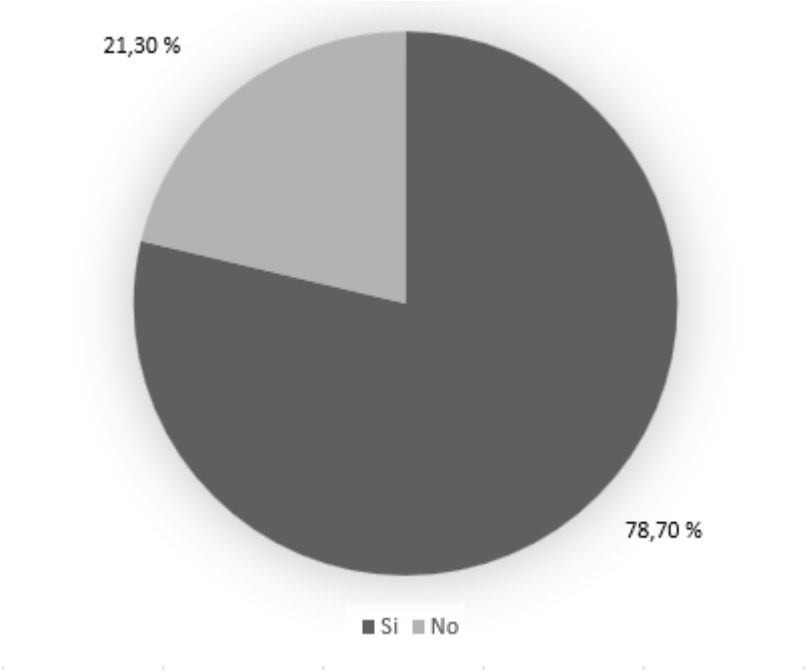
La descripción brindada era la siguiente: “La ingeniería del caos es la disciplina de experimentar en un sistema, con la finalidad generar confianza en la capacidad del sistema para soportar condiciones turbulentas en producción, es decir, realizar pruebas sobre nuestros sistemas en forma de experimentación para llevarlo al límite, de esta forma poder verificar la forma en la que los distintos componentes reaccionan a condiciones inesperadas para mejorar el sistema en la mayor medida posible, estos componentes pueden ser desde la red, contenedores, sistema operativo hasta el control de reloj interno de cada instancia” .[11] Tomada del marco teórico de esta misma tesis.

5.2.1. Pregunta 1

El 78,7 % indico no haber escuchado sobre *chaos engineering* previamente, siendo esto el equivalente a 37 de 47 personas que realizaron la encuesta. Esto muestra la poca popularidad que esta disciplina posee en Guatemala y el amplio campo que posee para explorar e implementar esta disciplina en la cultura informática moderna.

[11] COMMUNITY, Chaos. *Principios de la Ingeniería del Caos*. https://principlesofchaos.org/ Consulta: 5 de marzo de 2021.

Figura 37. **Porcentaje de conocimiento sobre *chaos engineering***



Fuente: elaboración propia, empleando Google Forms.

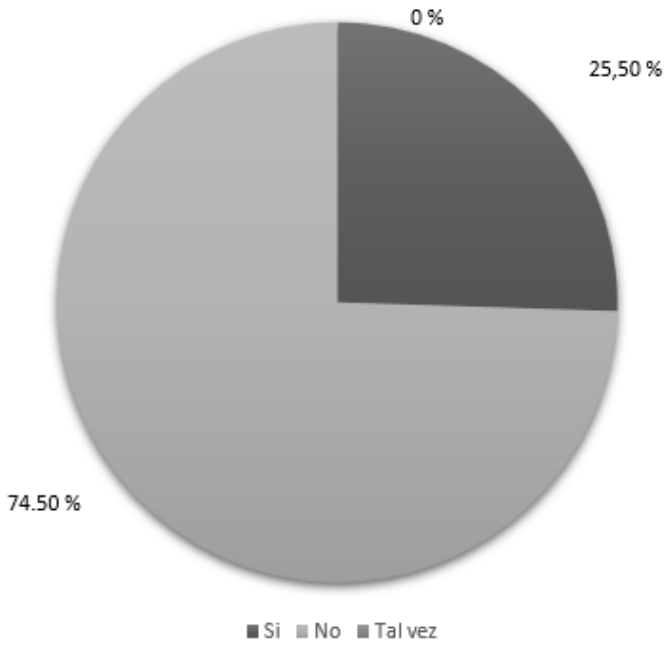
5.2.2. Pregunta 2

Un 25,5 % indico no estar seguro sobre si haya componentes dentro de un sistema informático que sea imposible que fallen mientras que un 74,5 % indico que no cree que haya componentes aprueba de fallos, mientras que nadie indico tener la seguridad de que haya componentes informáticos que sea imposible que fallen.

La utilidad de *chaos engineering* es tener en cuenta que no hay ningún componente dentro de un sistema que sea aprueba de fallas, se puede minimizar la cantidad de fallas o vulnerabilidades que posee, pero siempre hay que

considerar que puede fallar para su mejora continua, en este caso la mayoría tiene en mente esta consideración.

Figura 38. **Componentes de sistemas a prueba de fallas**



Fuente: elaboración propia, empleando Google Forms.

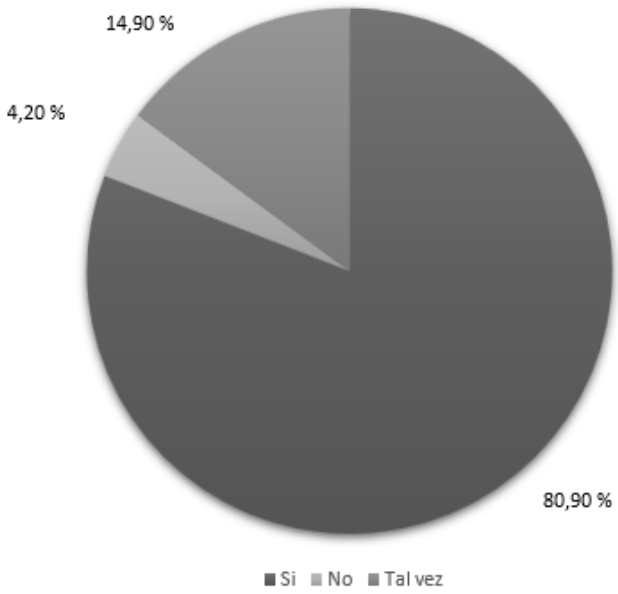
5.2.3. Pregunta 3

La tercera pregunta estaba condicionada a las personas que hubieran respondido en la pregunta 2 si consideraban que si hubiera componentes de un sistema informático en la nube que fuera imposible que fallaran. En este caso ninguna persona respondió afirmativamente.

5.2.4. Pregunta 4

Un 80,9 % de los participantes considera que, si es útil generar fallas a propósito dentro de un sistema informático que ya funciona correctamente, en contraste con un 14,9 % que no estaba seguro y un 4,20 % que indica que no lo considera útil. Considerando que chaos engineering busca provocar fallas para identificar puntos críticos y posibles vulnerabilidades una gran mayoría cree en el objetivo de esta disciplina.

Figura 39. **Generar fallas a propósito**

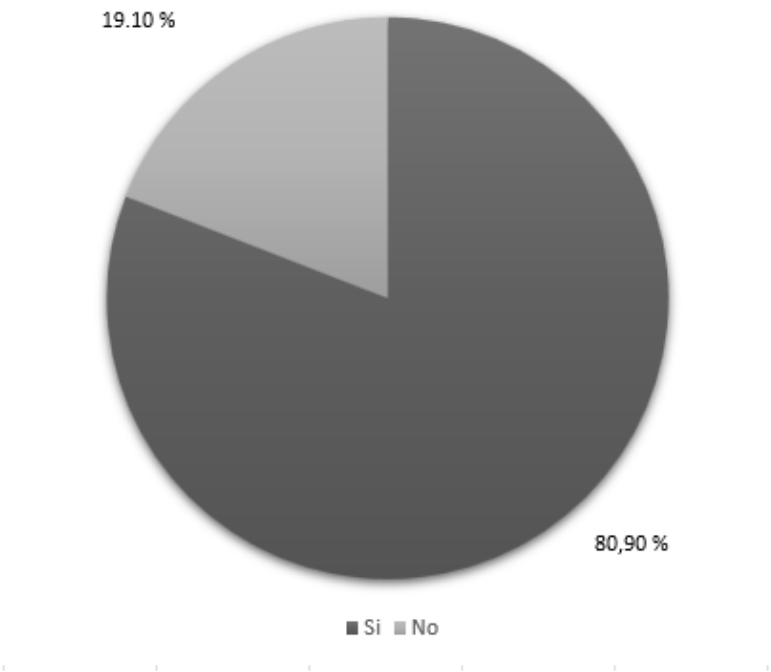


Fuente: elaboración propia, empleando Google Forms.

5.2.5. Pregunta 5

Un 80,9 % de los participantes desconoce en qué casos se debe aplicar *chaos engineering*, contra un 19,1 % que, si sabe, es decir que la mayoría de los participantes a pesar de verlo como una práctica importante desconoce las situaciones en que aplicarlo. *chaos engineering* puede ser aplicado en cualquier caso en el que se desee mejorar el sistema, aumentar su resiliencia, y disminuir los tiempos fuera de servicio, por mencionar algunos de los más importantes.

Figura 40. **Aplicación de *chaos engineering***

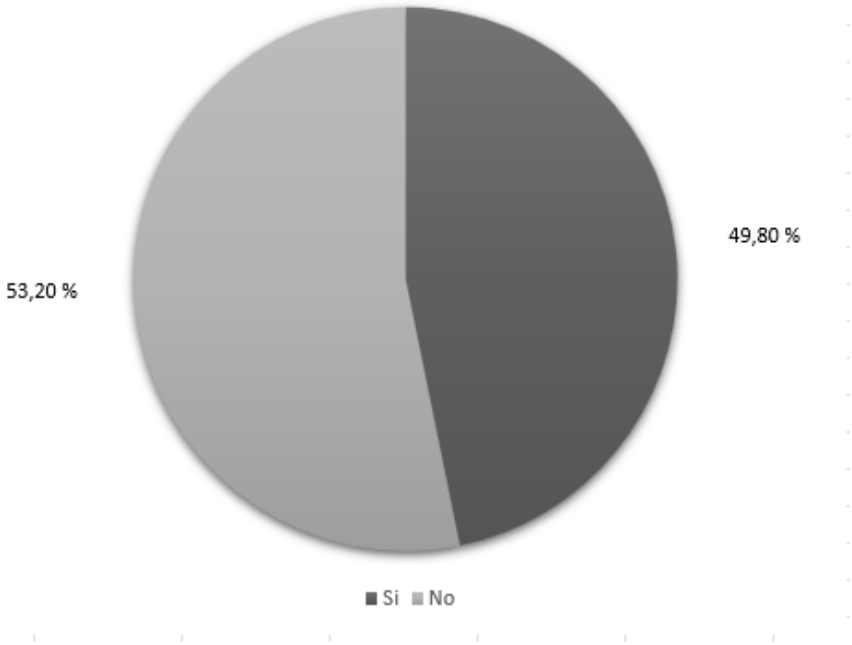


Fuente: elaboración propia, empleando Google Forms.

5.2.6. Pregunta 6

El 53,2 % de los participantes no conoce la finalidad de uso de *chaos engineering* en contraste con el 46,8 % que sí. Esto quiere decir que, aunque crean útil la generación de errores para la mejora del sistema desconocen el concepto y uso de esta disciplina, aunque indirectamente ya tengan las pautas de su utilidad y funcionamiento.

Figura 41. **Finalidad de uso de *chaos engineering***



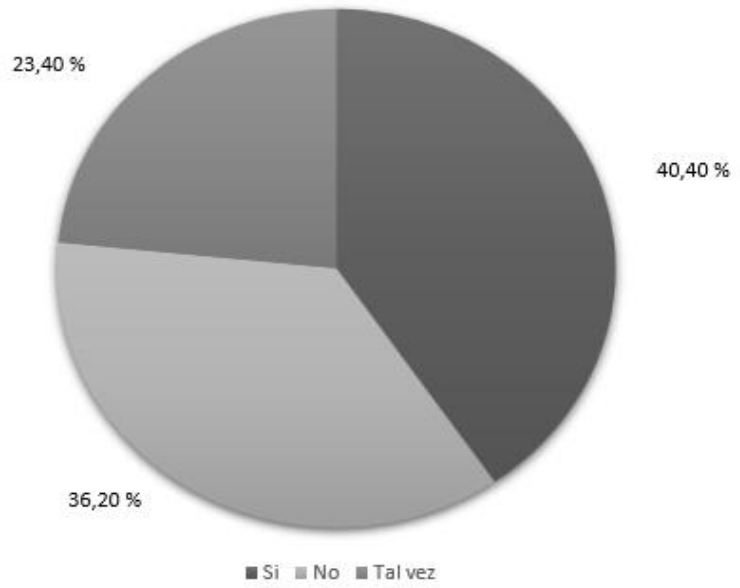
Fuente: elaboración propia, empleando Google Forms.

5.2.7. Pregunta 7

Con la pregunta 7 se contrasta de mejor forma la diversidad de opiniones acerca de sobre esforzar una arquitectura informática. El 40,4 % de los participantes indica si conocer que puede pasar en caso la arquitectura sobre pase su carga normal, un 36,2 % indica no conocer lo que puede llegar a pasar mientras que un 23,4 % indica no estar seguro de lo que pueda pasar.

Esto se debe a que muchos profesionales al tener una arquitectura funcional temen que se produzca una falla o no ha experimentado una falla en producción, es necesario conocer los posibles resultados que puedan darse al exponer cualquier arquitectura a condiciones extremas dado que no está exenta de que suceda, realizando esto se puede tomar medidas correctivas antes de que se produzca una falla en producción.

Figura 42. **Esfuerzo de una arquitectura informática**

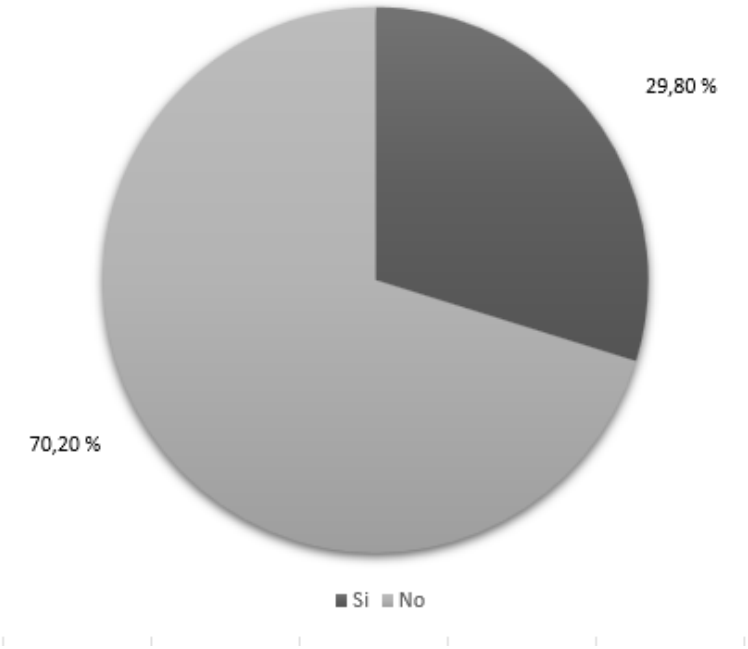


Fuente: elaboración propia, empleando Google Forms.

5.2.8. Pregunta 8

Un 70,2 % de los participantes indico no haber monitoreado lo que ocurre con una arquitectura informática cuando alguno de sus componentes fallas, en contraste con un 29,8 % que indica si haberlo hecho. Esto quiere decir que, al ocurrir una falla, al no tener un monitoreo constante, esta es detectada únicamente hasta que alguna funcionalidad critica colapsa, en lugar de mantener un monitoreo constante para identificar cualquier inconsistencia en el funcionamiento antes de que una falla produzca una caída de mayor grado.

Figura 43. **Monitoreo de una arquitectura informática**



Fuente: elaboración propia, empleando Google Forms.

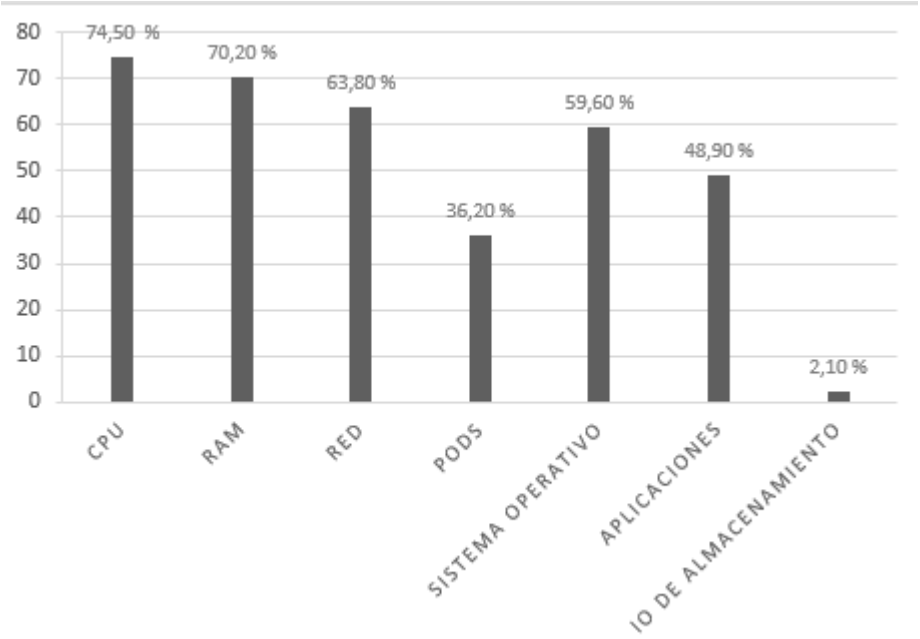
5.2.9. Pregunta 9

La pregunta 9 consiste en la selección múltiple de los componentes sobre los cuales a los participantes les gustaría realizar pruebas para poder asegurar su funcionamiento ante condiciones aleatorias, cada participante podría seleccionar una o más opciones por lo que cada opción es un porcentaje acumulado de la cantidad de usuarios que selecciono cada opción individual.

Un 74,5 % y 70,20 % seleccionaron que les gustaría realizar pruebas sobre el CPU y la RAM respectivamente para asegurar su funcionamiento, aunque en el manejo de entornos *cloud native* gracias al uso de Kubernetes y *auto scaling* dentro de los entornos estos dos componentes, que aunque eran críticos en arquitecturas *on premises*, ahora no debido a que en el caso de llegar a cierto límite en la utilización de estos componentes se agregan más replicas para balancear la carga de trabajo de las instancias evitando en la mayor medida posible una saturación.

En las arquitecturas modernas dirigidas hacia *cloud native* los nuevos componentes críticos se vuelven el funcionamiento general de los *pods* y la red de la arquitectura dado que se busca descentralizar el funcionamiento de esta para distribuir la carga.

Figura 44. **Selección de componentes para pruebas**

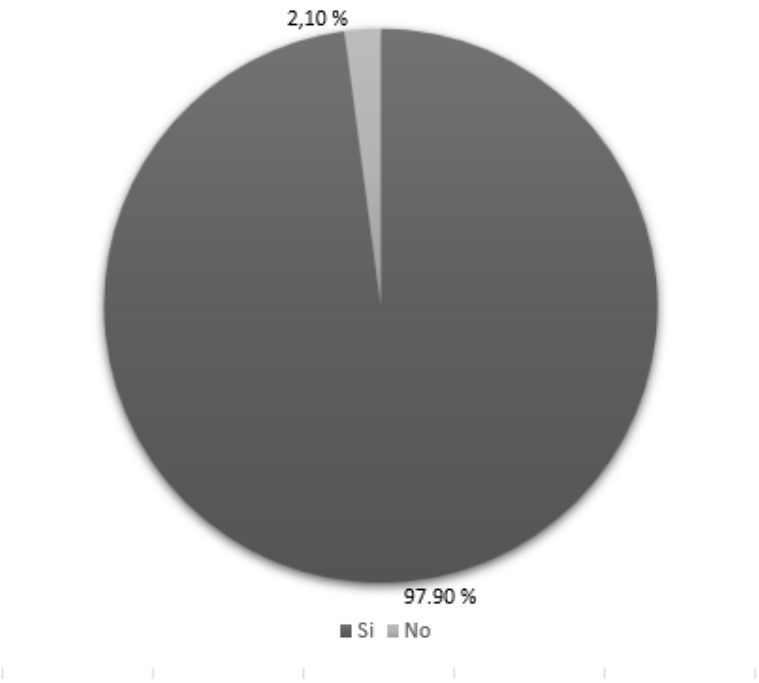


Fuente: elaboración propia, empleando Google Forms.

5.2.10. Pregunta 10

El 97,9 % indico que de existir herramientas especializadas en realizar pruebas a distintos componentes en el sistema están dispuestos a utilizarlas con la finalidad de mejorar la tolerancia a fallos en sus sistemas, en contraste con un 2,1 % que indico que no. En este caso se observa un interés por parte de la mayoría de los participantes en aplicar estas herramientas o *chaos engineering* para la mejora de sus sistemas.

Figura 45. **Uso de herramientas**

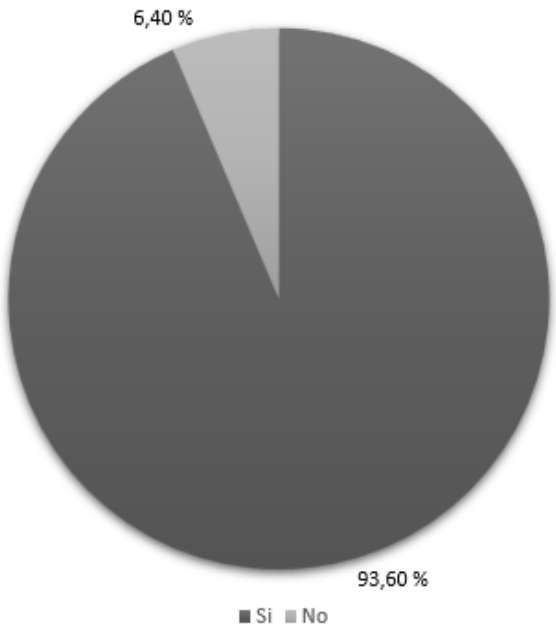


Fuente: elaboración propia, empleando Google Forms.

5.2.11. Pregunta 11

Un 93,6 % indico preferir usar herramientas *open source* para esta finalidad en contraste con un 6,4 % que indico preferir las herramientas de paga. Esto quiere decir que la mayoría está dispuesta a poder contribuir y experimentar con el uso de estas herramientas, las cuales actualmente la mayoría avaladas por CNCF y utilizadas en la industria son las herramientas *open source* para la aplicación de *chaos engineering*.

Figura 46. **Open source vs., pago**



Fuente: elaboración propia, empleando Google Forms.

5.3. Resumen de resultados

En base a los resultados obtenidos a través de las 11 preguntas realizadas en la encuesta, se obtuvo que la mayoría de las personas que realizaron la encuesta al momento de iniciar desconocían por completo sobre *chaos engineering* y los beneficios que trae la implementación de esta disciplina para la resiliencia del sistema.

En los resultados obtenidos se muestra que la mayoría de las personas que realizaron la encuesta desea poder realizar pruebas sobre los demás componentes de una arquitectura, no solo sobre la aplicación final, siendo de los componentes donde se obtuvo más intentes para pruebas el CPU, memoria

RAM, la red del sistema y el mismo sistema operativo sobre el cual se tienen ejecutando los diferentes servicios que se prestan. Buscan tener herramientas especializadas para este tipo de pruebas.

Se noto una diversidad de resultados en cuanto a conocer que puede ocurrir cuando se lleva una arquitectura o sistema al límite, una cantidad considerablemente pequeña de personas indico que ha realizado monitoreo sobre los sistemas, es decir que no están conscientes de como monitorear correctamente o de alguna forma cualquier sistema, por lo cual tampoco conocen las diferentes reacciones que podría tener el sistema.

Finalmente se muestra la preferencia del uso que se tendría sobre las herramientas, la cual se inclina sobre las herramientas *open source*, donde actualmente se desarrollan la mayoría de los proyectos y herramientas de alto nivel para *chaos engineering*. Con la distribución de la información mostrada se puede brindar una guía e introducción de como empezar a aplicar esta disciplina para cualquier interesado en su implementación.

5.4. Experiencia en ServiceMeshCon

Para la convocatoria de ServiceMeshCon Europe 2021 se envió la propuesta de charla con el tema central en la presente tesis, donde se mostraron 3 experimentos y los objetivos de la tesis, entre lo cual se presentó la información ante diversos espectadores.

Durante la conferencia los asistentes se mostraron interesados en el tema ya que a nivel mundial *chaos engineering* aún es una disciplina joven que se empieza a explorar y buscar sacar el mayor provecho de lo misma, incluso a nivel internacional los profesionales de informática se encuentran comenzando a

mostrar interés para aplicarla en diferentes organizaciones, pidiendo consejos de como empezar a experimentar e implementar *chaos engineering*, para lo cual la presentación les sirvió como una guía y ejemplo para esta labor.

CONCLUSIONES

1. Se encontró que en la actualidad *chaos engineering* tuvo sus inicios bajo el impulso y necesidad de Netflix para realizar pruebas sobre sus servicios mejorando considerablemente la resiliencia y estabilidad de estos, ahora es una práctica en crecimiento y que está siendo adoptada por una gran cantidad de empresas, evolucionando desde un sistema monolítico a través de un desglose abstracto hacia servicios, hasta el *service mesh* usado en el presente.

2. Se especifico el concepto de *chaos engineering* y se detalló la historia para la formación de esta disciplina, desde sus inicios con los experimentos de Netflix a inicios de la década pasada, hasta el momento en el que se definió específicamente esta disciplina en paralelo a la evolución de los sistemas desde un origen monolítico a través de un desglose de servicios, microservicios hasta el *service mesh*.

3. Se definieron los 5 principios asociados y aceptados como un estándar para *chaos engineering*, dado que nos permiten tener un camino a grandes rasgos sobre el cual podemos definir los experimentos, desde la generación de una hipótesis para los experimentos, hacia que experimentos tomar la prioridad de ejecución, la ejecución de los experimentos en entornos reales, automatizar estos experimentos hasta poder minimizar el radio de impacto de las fallas potenciales en el sistema para aumentar la resiliencia.

4. Se definieron distintas herramientas para la ejecución de experimentos de *chaos engineering* como Linkerd, Chaos Mesh, Chaos Monkey, Litmus, Latency Monkey, Gremlin Inc., entre otros. De igual forma se mostraron los diferentes experimentos que cada una de estas herramientas proporciona mostrando que Chaos Mesh y Linkerd son las dos herramientas que más experimentos poseen, ambas muy robustas y en base a esto se seleccionó Chaos Mesh para la ejecución de los experimentos dado que su curva de aprendizaje es relativamente pequeña y su implementación en conjunto con Linkerd para monitoreo en Kubernetes lo hicieron el candidato perfecto.

5. Se compararon las ventajas y desventajas de *chaos engineering*, siendo estas permitirnos identificar múltiples vulnerabilidades y fortalecer los sistemas contra un riesgo bajo de generar un problema por la inyección de un experimento no controlado. De igual forma se pudo observar durante el desarrollo de los experimentos que con cada uno se cumplía la base del caos, siendo esta no poder predecir completamente los resultados, cada experimento generaba fallas distintas a lo esperado permitiéndonos comparar el sistema desde un estado saludable a través de diferentes fallas que permitieron encontrar vulnerabilidades como la baja resiliencia del sistema para su mejora.

6. Los costos asociados a la implementación de *chaos engineering* dependerán de la herramienta que se utilice, es decir con la adopción de herramientas *open source* no habrá ningún costo asociado, dado que son de uso libre no se debe pagar por ninguna licencia, pero generara mejores resultados en cuanto al rendimiento y tolerancia a fallas del sistema minimizando la cantidad de vulnerabilidades potenciales que se pueden tener.

RECOMENDACIONES

1. Iniciar con la implementación de *chaos engineering* en cualquier entorno, sin tener conocimientos previos utilizar una arquitectura de prueba para no generar daños en producción. Definir desde el inicio los componentes que se consideran críticos dentro del sistema y su estado saludable para poder definir los tipos de experimentos que se deseen realizar sobre estos componentes, así mismo definir una hipótesis previa a la ejecución del experimento para tener la noción de los resultados que se esperan obtener al final. Durante la ejecución de los experimentos verificar en tiempo real utilizando herramientas que permitan tener observabilidad sobre el sistema, como se altera el comportamiento de este para poder tomar una decisión sobre si el sistema resiste el ataque para aumentar el radio de impacto o si es necesario mejorar el sistema.

2. Aplicar diferentes tipos de condiciones y tiempos de fallas sobre los diferentes experimentos que se realicen para verificar el comportamiento del sistema cuando este se somete a un mismo experimento en repetidas ocasiones, pero con diferentes niveles de tiempos de impacto para tener una visión más amplia sobre como cualquier falla podría afectar el sistema.

3. Realizar primero experimentos individuales sobre el sistema, es decir, un experimento sobre un *pod*, inyección de latencia, perdida de paquetes, entre otros, en solitario para poder verificar la forma en la que cada componente se ve afectado por una sola falla tomando en cuenta una forma ideal de vulnerabilidades. Posteriormente realizar un experimento que incluya todos los que previamente se han realizado o un grupo de experimentos seleccionados para poder contrastar la forma en la que el sistema se comporta con múltiples fallas ocurriendo al mismo tiempo sobre él.

BIBLIOGRAFÍA

1. Aplica. *Service mesh: Arquitectura de Microservicios*. [en línea]. <https://www.aplyca.com/es/blog/service-mesh>. [Consulta: 7 de febrero 2021].

2. BURNS, Brendan. *Designing Distributed Systems*. 1a ed. Estados Unidos de América: O'Reilly Media. 2018. 168 p.

3. CAREY, Scott. *¿Qué es Chaos Monkey? La Ingeniería del caos, explicada*. [en línea]. <https://cioperu.pe/articulo/30319/que-es-chaos-monkey-la-ingenieria-del-caos-explicada/?p=3>. [Consulta: 4 de febrero 2021].

4. COMMUNITY, Chaos. *Principles Of Chaos Engineering*. [en línea]. <https://principlesofchaos.org/>. [Consulta: 4 de febrero 2021].

5. Google Cloud. *Instala Google Cloud CLI*. [en línea]. <https://cloud.google.com/sdk/docs/install-sdk>. [Consulta: 20 de febrero de 2021].

6. NUÑEZ, Angel. *Chaos Engineering*. [en línea]. <https://es.slideshare.net/snahider/chaos-engineering-168238678>. [Consulta: 6 de febrero 2021].

7. ROSENTHAL, Casey; JONES, Nora. Chaos Engineering. 1a ed. Estados Unidos de América: O'Reilly Media, 2020. 308 p.

8. SECURITY, Panda. *Ingeniería del Caos: ¿Para qué sirve introducir fallos adrede?* [en línea]. <https://www.pandasecurity.com/es/mediacenter/seguridad/ingenieria-del-caos-para-que-sirve-introducir-fallos-adrede/>. [Consulta: 4 de febrero 2021].

9. SUTTER, Burr; POSTA, Christian. *Introducing Istio Service mesh for Microservices*. 2a ed. Estados Unidos de América: O'Reilly Media, 2019. 85 p.

10. VELIMIROVIC, Andreja. *Chaos Engineering: How it Works, Principles, Benefits & Tools*. [en línea]. <https://phoenixnap.com/blog/chaos-engineering#:~:text=Chaos%20engineering%20helps%20stop%20large,the%20reliability%20of%20their%20services>. [Consulta: 10 de febrero 2021].

APÉNDICES

Apendice 1. **Grabación de evento ServiceMeshCon 2021**

Fuente: elaboración propia, empleando captura de pantalla.

- Enlace de video en canal oficial de CNCF con la presentación del tema de tesis como parte del evento ServiceMeshCon Europe 2021: https://www.youtube.com/watch?v=vGVtnP8gOl8&list=PLj6h78yzYM2Pi4GsjNNWkkEvntCLMTjEL&index=20[12]

[12] The Linux Foundation. *Creating chaos in the university with Linkerd and Chaos Mesh J.B.C. Fajardo & S.A.M. Aguilar*. https://www.youtube.com/watch?v=vGVtnP8gOl8&list=PLj6h78yzYM2Pi4GsjNNWkkEvntCLMTjEL&index=20. Consulta: 15 de mayo de 2021.

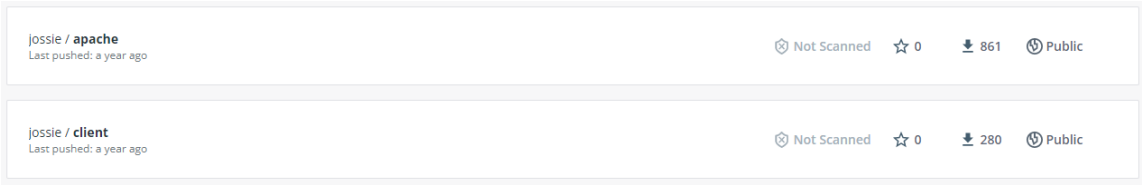
Apendice 2.**Repositorio de código**

Fuente: elaboración propia, empleando captura de pantalla.

- Repositorio de código para replicar los experimentos.

  https://github.com/jossiebk/ExperimentosTesis

Apendice 3. **Aplicaciones contenerizadas**



Fuente: elaboración propia, empleando captura de pantalla.

- Repositorio de Docker Hub con las imágenes generadas.
  - Servidor: https://hub.docker.com/repository/docker/jossie/Apache
  - Cliente: https://hub.docker.com/repository/docker/jossie/client

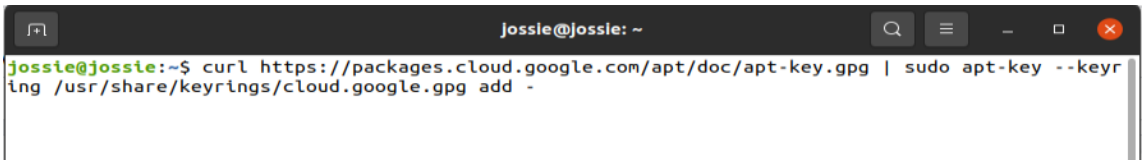
Apendice 4. **Paso 1. Agregar SDK de Google Cloud como recurso de paquetes en el sistema**

Fuente: elaboración propia, empleando captura de pantalla.

- Paso 1: agregar la dirección del SDK de Google Cloud como un recurso de paquetes dentro del sistema.

  ~$ echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] http://packages.cloud.google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/sources.list.d/google-cloud-sdk.list

Apendice 5. **Paso 2. Importar la llave publica de GCP**

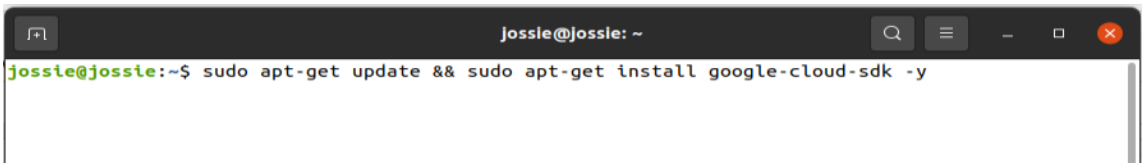


Fuente: elaboración propia, empleando captura de pantalla.

- Paso 2: importar la llave publica de Google dentro del entorno.

  ~$ curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key --keyring /usr/share/keyrings/cloud.google.gpg add –

Apendice 6. **Paso 3. Actualizar paquetes e instalar CGP SDK**

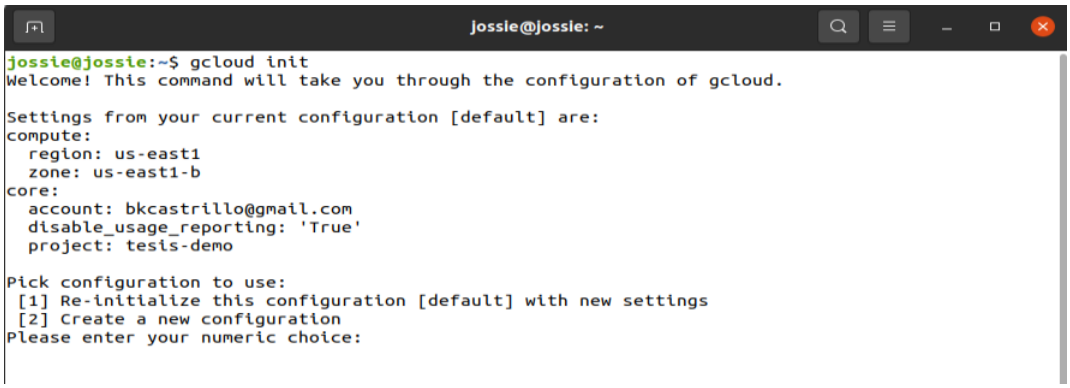


Fuente: elaboración propia, empleando captura de pantalla.

- Paso 3: actualizar la lista de paquetes e instalar Cloud SDK, tomar en cuenta que esta instalación puede tardar varios minutos.

  ~$ sudo apt-get update && sudo apt-get install google-cloud-sdk

Apendice 7.**Paso 4. Inicializar SDK**

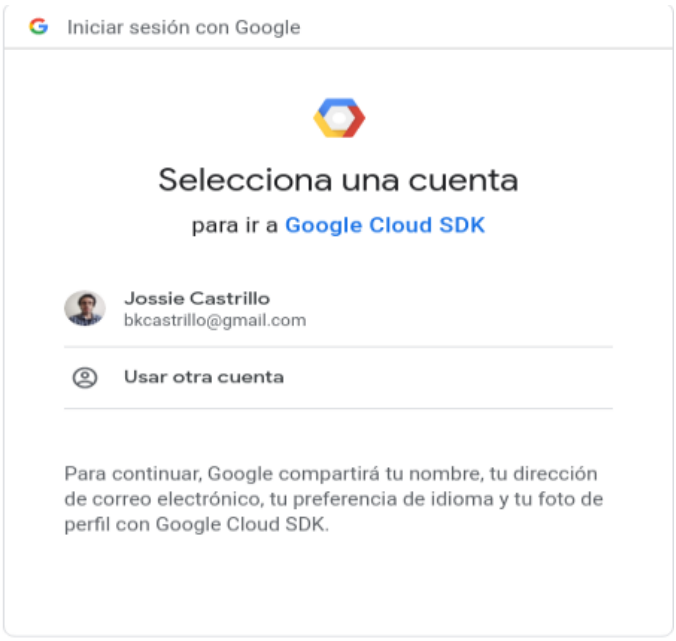


Fuente: elaboración propia, empleando captura de pantalla.

- Paso 4: inicializar el SDK, luego de algunos segundos pedirá iniciar sesión con una cuenta para lo cual se debe presionar “Y” y abrirá una ventana en el navegador para ingresar con la cuenta deseada.

  ~$ gcloud init

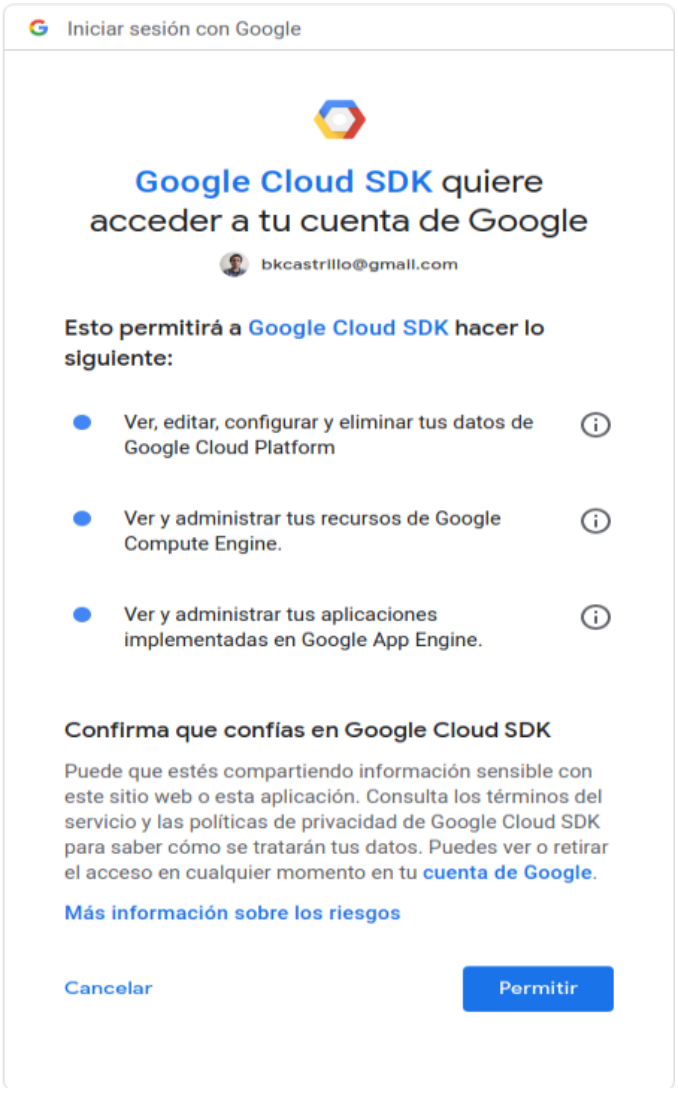
Apendice 8. **Paso 4.1. Inicio de sesión en GCP**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 4.1: seleccionar una cuenta para iniciar sesión.

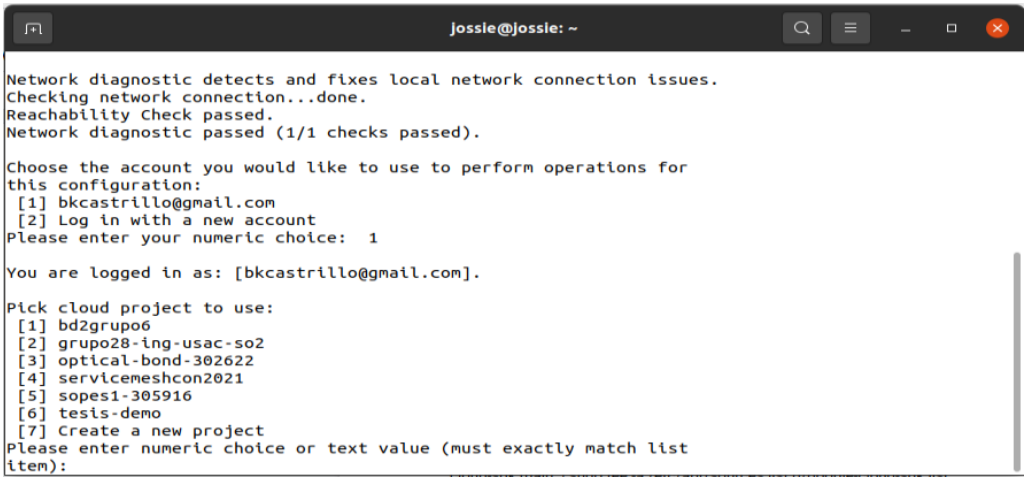
Apendice 9. **Paso 4.2. Permitir acceso de Google Cloud SDK**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 4.2: se permite el acceso y procederá a mostrar una ventana en la cual indica que ya se ha accedido.

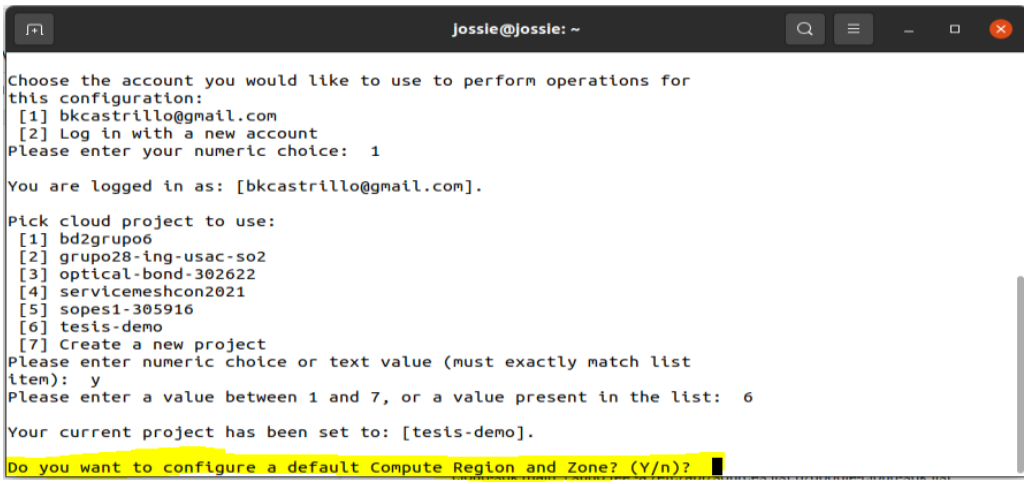
Apendice 10. **Paso 5. Selección de proyecto**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 5: al terminar solicita seleccionar un proyecto previamente creado en Google Cloud, se seleccionó indicando el numero respectivo a su nombre, 6 haciendo referencia al proyecto tesis-demo.

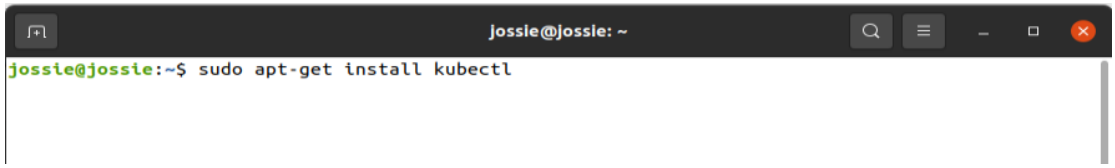
Apendice 11. **Paso 6. Selección de región y zona**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 6: se selecciona una región y zona por defecto para trabajar y se presiona “Y”.

Apendice 12. **Paso 7. Instalación de cliente de Kubernetes**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 7: instalar el cliente de Kubernetes.

  ~$ sudo apt-get install kubectl

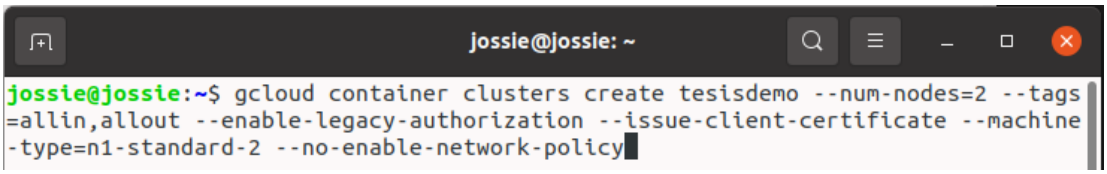
Apendice 13. **Paso 8. Verificación de instalación de *kubectl***

Fuente: elaboración propia, empleando captura de pantalla.

- Paso 8: verificar la instalación.

  ~$ kubectl version --short

Apendice 14. **Paso 9. Crear clúster de Kubernetes**



Fuente: elaboración propia, empleando captura de pantalla.

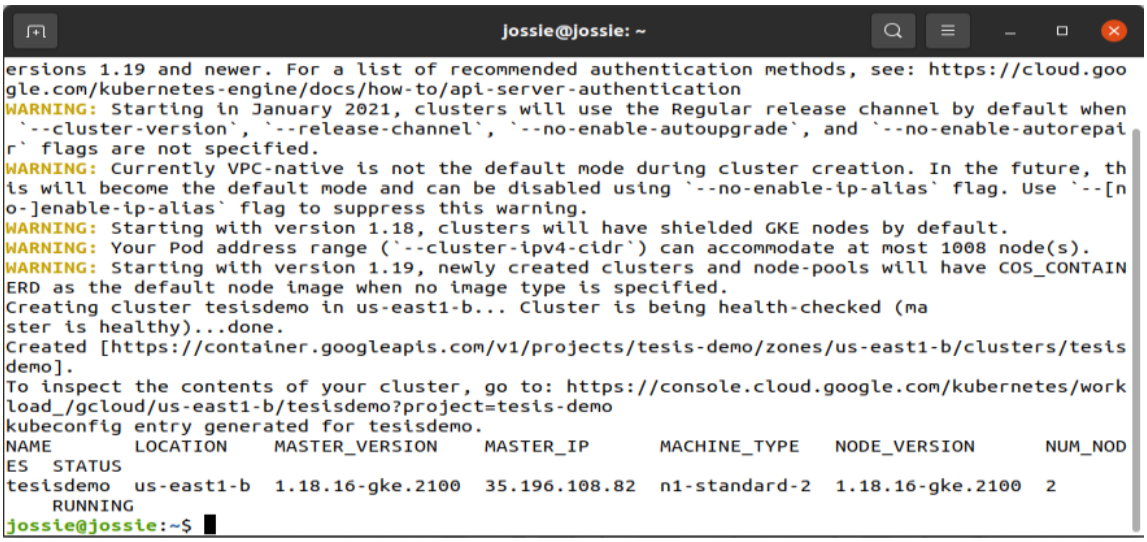
- Paso 9: crear un clúster para el despliegue de la arquitectura.

  ~$ gcloud container clusters create tesisdemo --num-nodes=2 --tags=allin,allout --enable-legacy-authorization --issue-client-certificate --machine-type=n1-standard-2 --no-enable-network-policy

  Explicación de banderas utilizadas:

  - --num-nodes=2: esta bandera sirve para definir la cantidad de nodos que tendrá el clúster, en este caso es generaron únicamente 2.
  - --tags=allin,allout: se indican *tags* referentes a reglas de *firewall* con la apertura de puertos, en este caso se permitió todo el tráfico de entrada o salida.
  - --enable-legacy-authorization: esta política de Kubernetes otorga permisos definidos a todos los usuarios del clúster.
  - --machine-type=n1-standard-2: esta etiqueta sirve para definir el tipo de *hardware* que se utilizara, el tipo n1 standard 2 es de uso general el cual posee 2 núcleos, 7,50Gb de memoria RAM, un máximo de 128 discos persistentes, un almacenamiento local de 257TB en SSD y un ancho de banda de 10Gbps.

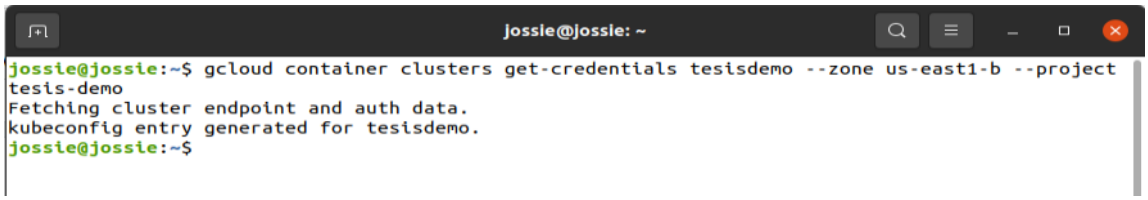
Apendice 15. **Paso 9.1. Mensaje de estatus de clúster**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 9.1: se muestra un mensaje con el estatus final y se puede visualizar desde la plataforma en la nube el clúster creado.

Apendice 16. **Paso 10. Obtener credenciales de clúster**

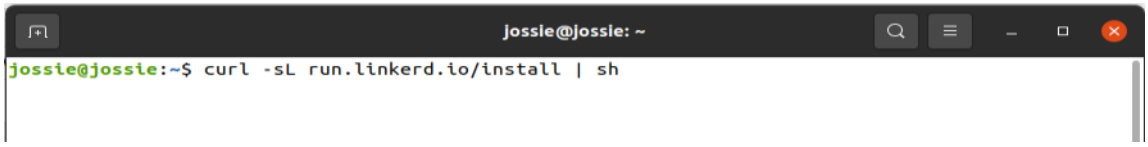


Fuente: elaboración propia, empleando captura de pantalla.

- Paso 10: se obtienen las credenciales para el manejo del clúster.

  ~$ gcloud container *clusters* get-credentials tesisdemo --zone us-central1-c --project tesis-demo

Apendice 17. **Paso 11. Instalación de Linkerd**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 11: instalar la línea de comandos de Linkerd.

  ~$ curl -sL run.linkerd.io/install | sh

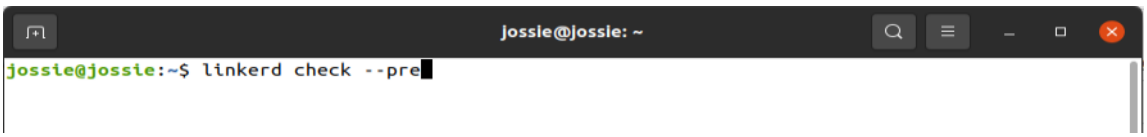
Apendice 18. **Paso 11.1. verificación de instalación de Linkerd**

Fuente: elaboración propia, empleando captura de pantalla.

- Paso 11.1: verificar instalación.

  ~$ linkerd version

Apendice 19. **Paso 12. Validar requerimientos para Linkerd**

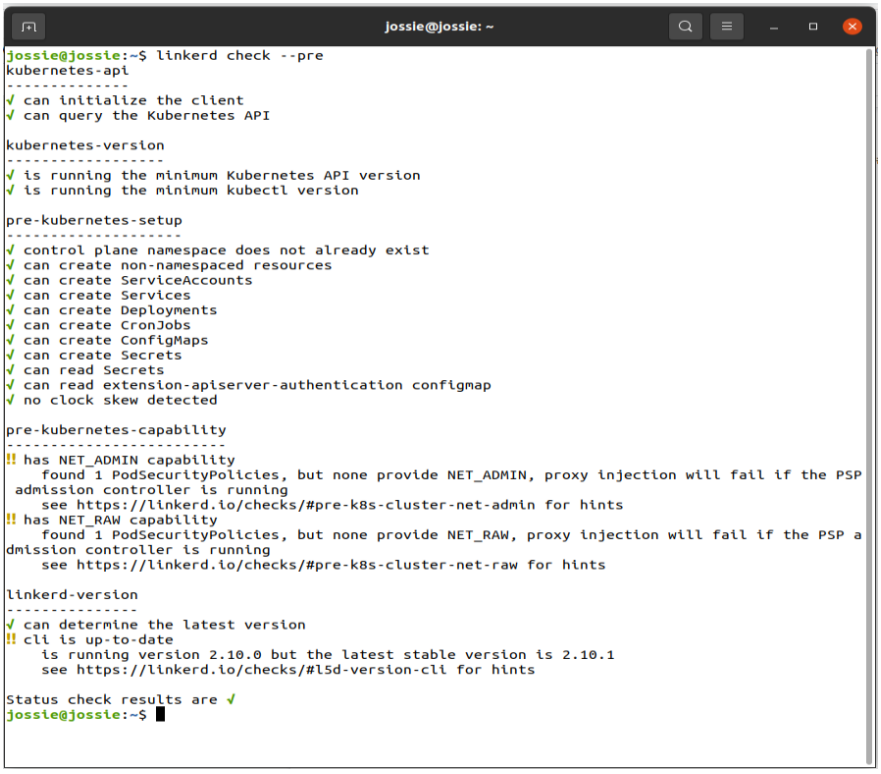


Fuente: elaboración propia, empleando captura de pantalla.

- Paso 12: validar que el clúster de Kubernetes creado cumple con lo necesario para Linkerd.

  ~$ linkerd check –pre

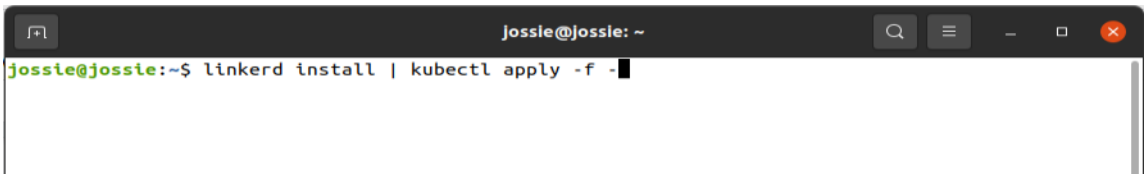
Apendice 20. **Paso 12.1. Validación de Linkerd**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 12.1: al estar todo correcto muestra si el clúster está listo para la instalación posterior.

Apendice 21. **Paso 13. Instalación de *control plane* en clúster**



Fuente: elaboración propia, empleando captura de pantalla.

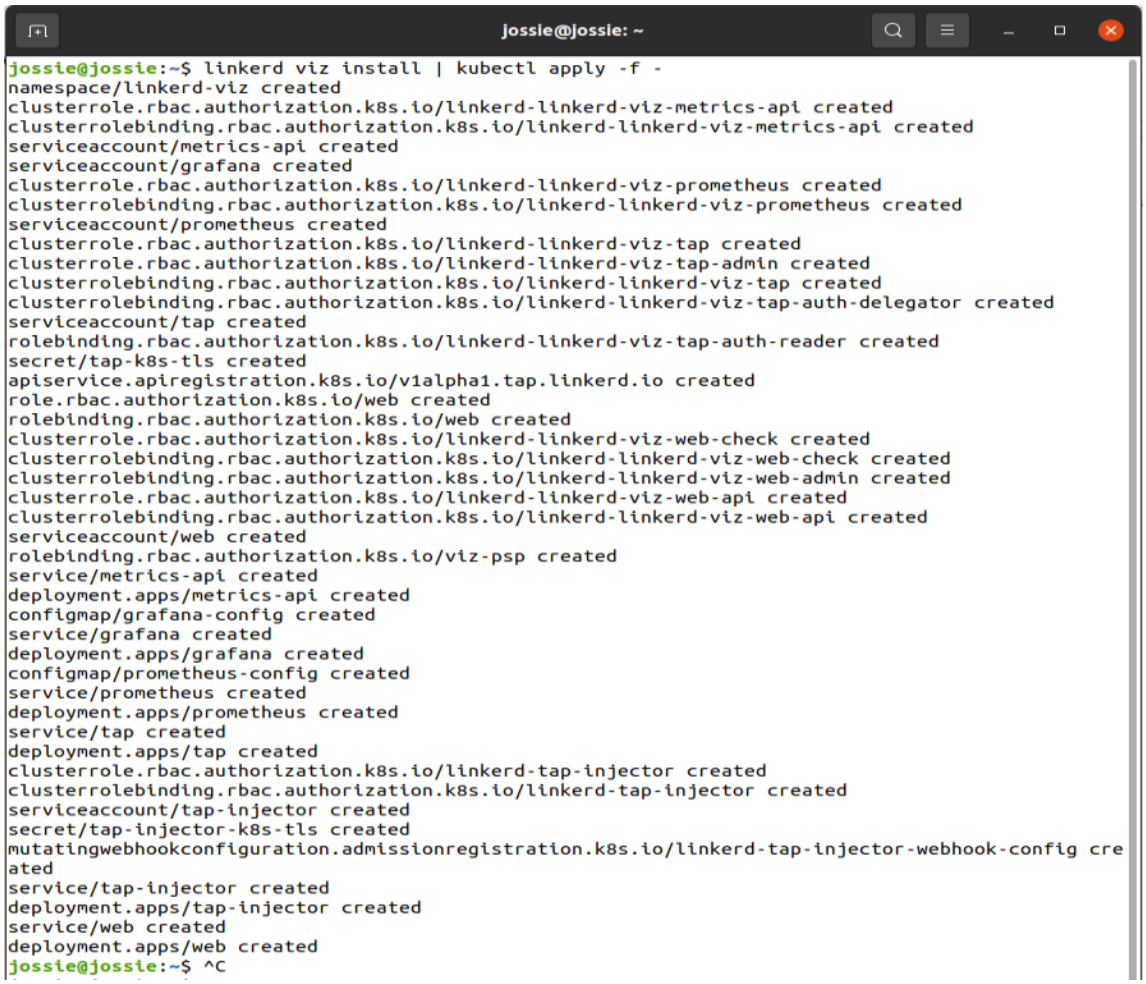
- Paso 13: instalar el *control plane* dentro del clúster, esto genera la observabilidad para el mismo. Tener en cuenta que esta configuración tarda varios minutos.

  ~$ linkerd install | kubectl apply -f –

- Paso 14: verificar la instalación.

  ~$ linkerd check

Apendice 22. **Paso 15. Instalar *dashboard* de Linkerd**



Fuente: elaboración propia, empleando captura de pantalla.

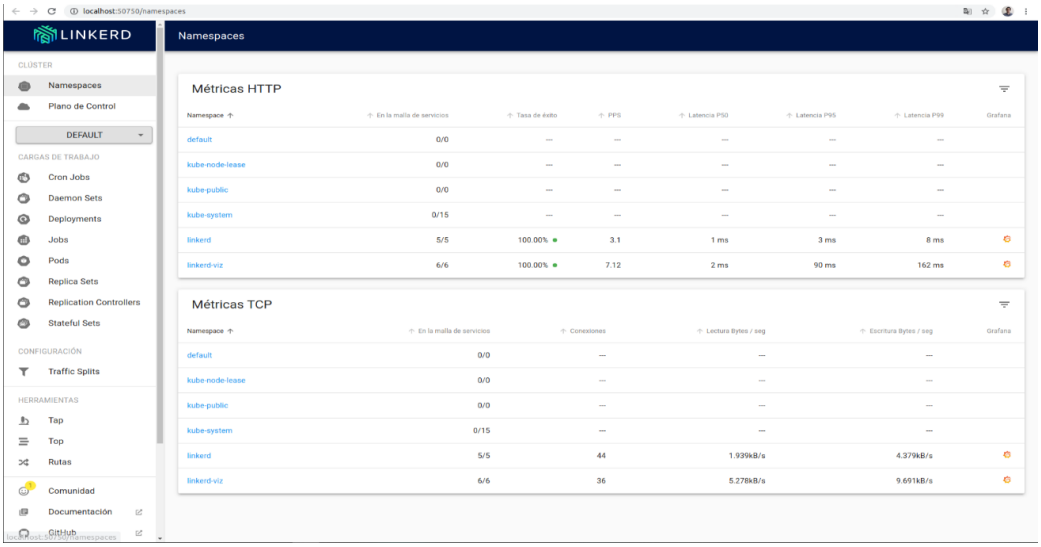
- Paso 15: agregar la extensión para visualizar el estado de la arquitectura desde el *dashboard* de Linkerd.

  ~$ linkerd viz install | kubectl apply -f –

Apendice 23. **Paso 16. Habilitar *dashboard* de Linkerd**

Fuente: elaboración propia, empleando captura de pantalla.

Apendice 24. **Vista preliminar del *dashboard* de Linkerd**



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 16: habilitar el *dashboard* cuando sea necesario para el monitoreo, esto abre una ventana en el navegador con el *dashboard*.

```
~$      linkerd viz dashboard &
```

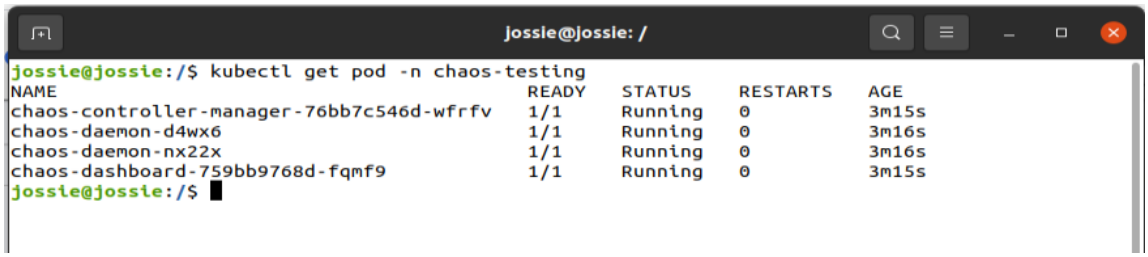
Apendice 25. **Paso 17. Instalación de *chaos mesh* en el sistema**

Fuente: elaboración propia, empleando captura de pantalla.

- Paso 17: instalar Chaos Mesh.

  ~$ curl -sSL https://mirrors.chaos-mesh.org/v2.2.0/install.sh | bash

Apendice 26. **Paso 18. Verificación de *chaos mesh***



Fuente: elaboración propia, empleando captura de pantalla.

- Paso 18: verificar instalación.

  ~$ kubectl get pod -n chaos-testing

Apendice 27. **Paso 19. Construir aplicaciones**

Fuente: elaboración propia, empleando captura de pantalla.

Para esta sección es necesario clonar el repositorio adjunto en los anexos dentro del sistema para los pasos posteriores y la generación de la aplicación de prueba.

- Paso 19: ingresar a la carpeta de la aplicación Cliente y ejecutar el siguiente comando.

  ~$ sudo ./build.sh jossie

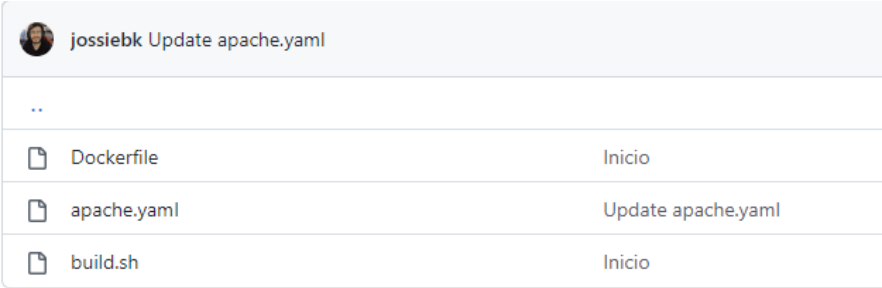
Apendice 28. **Paso 20. Construir imágenes de Docker**

Fuente: elaboración propia, empleando captura de pantalla.

- Paso 20: ingresar la contraseña de su Docker Hub e iniciara la construcción de la imagen, posteriormente se puede verificar en su repositorio la misma. Se puede utilizar las aplicaciones ya creadas en el repositorio de la plantilla de la arquitectura.

- Paso 21: repetir los dos pasos anteriores con la carpeta Servidor

Apendice 29. **Ejemplo de archivos de aplicación**



Fuente: elaboración propia, empleando captura de pantalla.

La totalidad de la arquitectura esta implementada dentro de un archivo con extensión Yaml para un despliegue sencillo en el cual se definen ambas aplicaciones

Apendice 30. **Código de aplicación cliente**

```
1  FROM ubuntu:18.04
2  MAINTAINER Jossie
3  RUN apt-get update
4  RUN apt-get install -y siege
5  CMD ["siege","-b","http://apache"]
```

Fuente: elaboración propia, empleando captura de pantalla.

Para la definición de la aplicación cliente se definió un *dockerfile* en base al cual se generó una imagen para el despliegue utilizado como sistema Ubuntu 18.04 y una actualización del sistema. Posteriormente se instaló y ejecuto Siege, la cual es una herramienta orientada a las pruebas sobre peticiones http y https, para los fines de los experimentos estará realizando peticiones continuas a los servidores Apache.

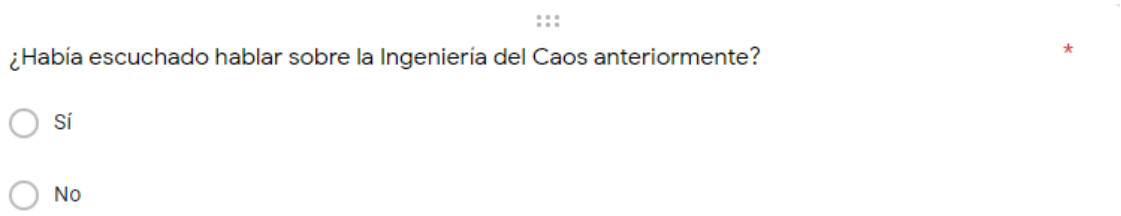
Apendice 31. **Código de aplicación servidor**

```
1  FROM ubuntu:18.04
2  MAINTAINER Jossie
3  RUN apt-get update
4  RUN apt-get install -y apache2
5  EXPOSE 80
6  CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Fuente: elaboración propia, empleando captura de pantalla.

Para la definición del servidor se utiliza una imagen de Ubuntu 18.04 en la cual se realiza una actualización del sistema y la instalación de Apache2 como servidor web. Posteriormente se expuso el servidor Apache en el puerto 80 del contenedor para las peticiones.

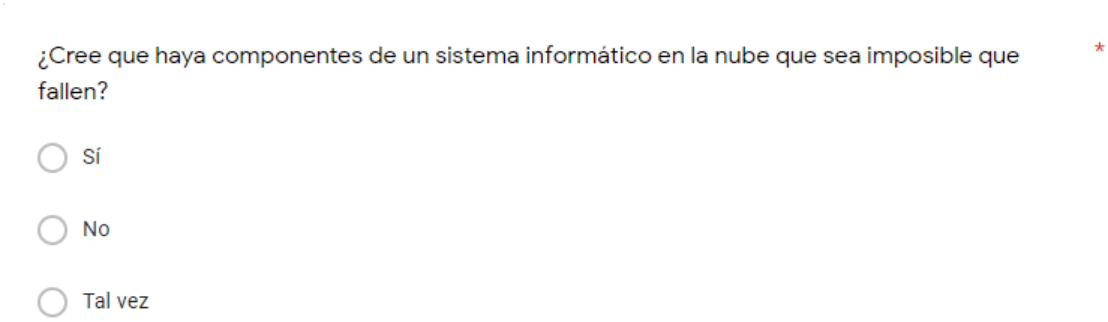
Apendice 32. **Pregunta 1. conocimiento de ingeniería de caos**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 1: ¿había escuchado hablar sobre la ingeniería del caos anteriormente?

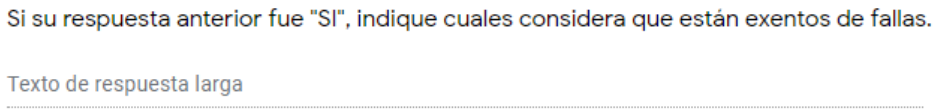
Apendice 33. **Pregunta 2. Conocimiento de componentes**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 2: ¿cree que haya componentes de un sistema informático en la nube que sea imposible que fallen?

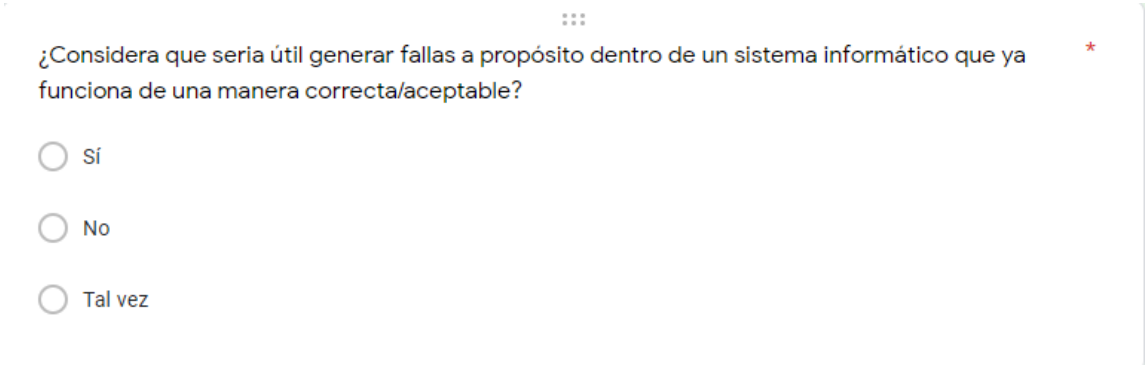
Apendice 34. **Pregunta 3. En base a la pregunta anterior**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 3: si su respuesta anterior fue “SI”, indique cuales considera que están exentos de fallas.

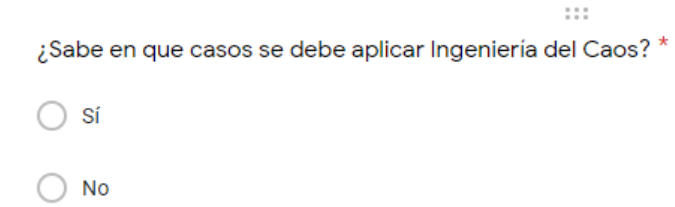
Apendice 35. **Pregunta 4. Utilidad de fallas a propósito**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 4: ¿considera que sería útil generar fallas a propósito dentro de un sistema informático que ya funciona de una manera correcta/aceptable?

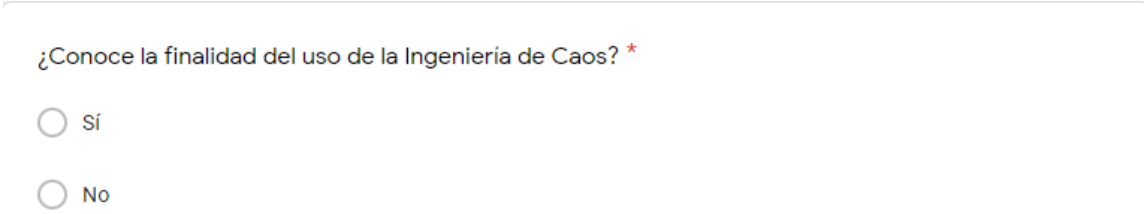
Apendice 36. **Pregunta 5. Aplicación de ingeniería de caos**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 5: ¿sabe en qué casos se debe aplicar ingeniería del caos?

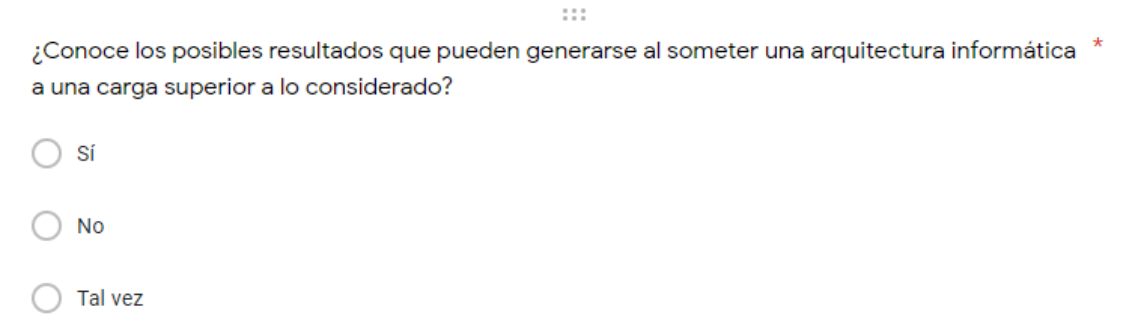
Apendice 37. **Pregunta 6. Finalidad de ingeniería de caos**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 6: ¿conoce la finalidad del uso de la ingeniería de caos?

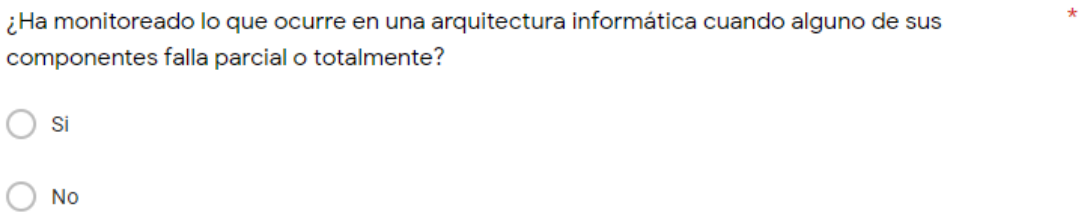
Apendice 38. **Pregunta 7. Posibles resultados**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 7: ¿conoce los posibles resultados que pueden generarse al someter una arquitectura informática a una carga superior a lo considerado?

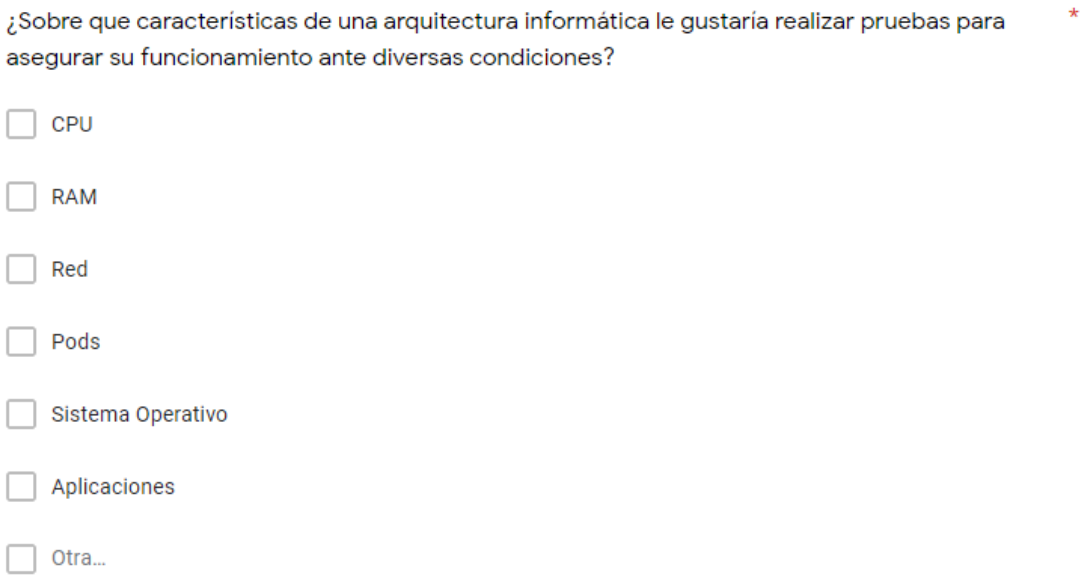
Apendice 39. **Pregunta 8. Monitoreo de arquitecturas**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 8: ¿ha monitoreado lo que ocurre en una arquitectura informática cuando alguno de sus componentes falla parcial o totalmente?

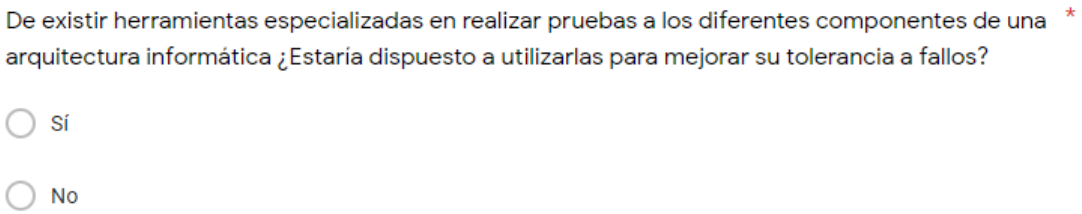
Apendice 40. **Pregunta 9. Características de arquitecturas**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 9: ¿Sobre qué características de una arquitectura informática le gustaría realizar pruebas para asegurar su funcionamiento ante diversas condiciones?

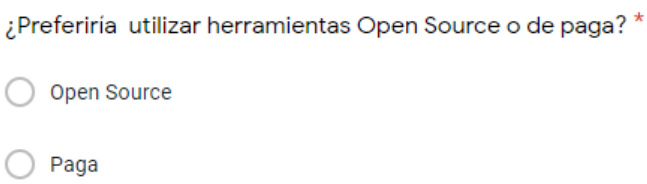
Apendice 41. **Pregunta 10. Herramientas para pruebas**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 10. De existir herramientas especializadas en realizar pruebas a los diferentes componentes de una arquitectura informática ¿Estaría dispuesto a utilizarlas para mejorar su tolerancia a fallos?

Apendice 42. **Pregunta 11. open source vs paga**



Fuente: elaboración propia, empleando captura de pantalla.

- Pregunta 11. ¿Preferiría utilizar herramientas *open source* o de paga?