Maestría Profesional en Ingeniería del Software

Documento Final de Proyecto de Investigación Aplicada 2

Caso de estudio: Aplicación de principios de antifragilidad en sistemas distribuidos basados en microservicios

Miranda Patiño Marco Vinicio

Febrero 2024

TRIBUNAL EXAMINADOR

Este proyecto fue aprobado por el Tribunal Examinador de la carrera: **Maestría Profesional en Ingeniería del Software con énfasis en Arquitectura y Diseño de Software,** requisito para optar por el título de grado de **Maestría,** para el estudiante: **Miranda Patiño Marco Vinicio.**

IGNACIO TREJOS ZELAYA (FIRMA)
Firmado digitalmente por IGNACIO TREJOS ZELAYA (FIRMA)
Fecha: 2024.03.18 22:31:59 -06'00'

M.Sc. Ignacio Trejos Zelaya
Tutor

JIRLEY RAMIREZ CORDERO (FIRMA)
Digitally signed by JIRLEY RAMIREZ CORDERO (FIRMA)
Date: 2024.03.19 08:15:07 -06'00'

M.Sc. Shirley Ramírez Cordero
Lector 1

FRANCISCO VARGAS NAVARRO (FIRMA)
PERSONA FISICA, CPF-09-0099-0713.
Fecha declarada: 20/03/2024 08:08:45 AM
Razón: FVNcr.org
Lugar: Costa Rica Contacto: Ing. Francisco Vargas

Máster Francisco Vargas Navarro
Lector 2

San José, Costa Rica, 7 de marzo de 2024

Firmada digitalmente, de conformidad con la Ley de Certificados, Firmas Digitales y Documentos Electrónicos N° 8454, destacando el artículo 9°-

Aplicación de principios de antifragilidad en sistemas distribuidos basados en microservicios

Marco Miranda
[#] *Escuela de Ingeniería del Software, Universidad CENFOTEC*
San José, Costa Rica
[1] mmirandap@ucenfotec.ac.cr

Resumen **—** La antifragilidad propone una nueva característica de sistemas de software resilientes en la que no solo son capaces de resistir, recuperarse y seguir funcionando en presencia de errores, sino también de mejorar al tomar el impacto como una oportunidad de aprendizaje. Este artículo explora los principios del software antifrágil, tanto en las propiedades del software como en los procesos utilizados en su construcción y mantenimiento, y propone una serie de mejoras para cada principio que se apliquen a un software empresarial existente basado en microservicios, con el propósito de alcanzar un mayor nivel de madurez apuntando a la antifragilidad.

Palabras clave: Antifragilidad, resiliencia, tolerancia a fallos, inyección de fallos

Abstract **—** Antifragility proposes a new characteristic of resilient software systems where they are not only able to resist, recover and continue working in the presence of errors, but also to improve by taking the impact as an opportunity of learning. This article goes through the principles of antifragile software - both in the properties of the software and in the processes used in how to build and maintain them - and proposes a series of improvements for each principle to be applied to an existing microservice-based enterprise software, with the purpose of achieving a better maturity level aiming at antifragility.

Keywords: Antifragility, resilience, fault-tolerance, fault-injection

I. Introducción

En el diseño e implementación de sistemas de software distribuidos modernos, es común encontrar la aplicación y uso de estilos arquitectónicos basados en microservicios. Su auge se debe principalmente a sus ventajas en escalabilidad independiente - aprovechando de manera más eficiente los recursos computacionales disponibles -, heterogeneidad tecnológica - los microservicios pueden emplear diferentes tecnologías entre sí que permiten resolver sus problemas concretos de una mejor manera -, facilidad de *deployment* (puede ser liberado de manera modular, con bajo impacto y riesgo), reutilización - un mismo microservicio puede ser utilizado por múltiples entidades -, reemplazo de componentes - un componente puede ser intercambiado por otro siempre y cuando cumpla la misma interfaz -, alineación organizacional - los equipos de ingeniería se pueden coordinar más eficientemente para trabajar en paralelo sobre múltiples microservicios en lugar de un solo proyecto -, así como resiliencia. [1] [12]

En el contexto de sistemas de software distribuidos, se define que "un sistema […] es resiliente cuando puede continuar realizando su trabajo aún con la presencia de fallas" [2]. Sin embargo, debido a la gran demanda por acceso a sistemas computacionales y al crecimiento de la complejidad en estos, así como por las expectativas de calidad que poseen los usuarios finales, continuar el funcionamiento en presencia de errores no solo es un trabajo cada vez más difícil, sino que además carece de capacidades de mejora ante la presencia de incidentes.

Este trabajo se inspira en un nuevo concepto llamado *antifragilidad*, donde un sistema no solo es capaz de resistir ante impactos, sino que además obtiene un aprendizaje resultando en una versión de mejor calidad. Se realiza el análisis de un sistema moderno basado en microservicios - considerado por el equipo de ingeniería encargado de este como uno de alta resiliencia - y se proponen posibles implementaciones que buscarían alcanzar un mayor nivel de madurez en lo referente a antifragilidad. Tal análisis se realiza utilizando como base los principios de software antifragilidad propuestos por *Martin Monperrus* en su artículo *Principles of Antifragile Software* [3].

II. Antifragilidad como nivel de madurez

El enfoque tradicional sobre resiliencia determina la capacidad de un sistema de continuar funcionando ante la posible presencia de errores o fallas, sin describir cuán bien lo hace, ni su estado final una vez el impacto del error ha mermado.

Asimismo, el conocimiento de niveles de madurez de un atributo de calidad en un sistema de software permite evaluar y determinar posibles puntos de mejora, debilidades existentes, y fortalezas que este posea.

El análisis del sistema de software distribuido basado en microservicios en este artículo se realiza mediante la propuesta de una secuencia de niveles de madurez relacionados a la resiliencia, los cuales se describen así:

- *Frágil:* Un sistema que ante la presencia de una falla o error (anteriormente conocido o no), presenta degradación del servicio a tal nivel que es incapaz de ofrecer valor a sus usuarios finales hasta ser intervenido manualmente. Un sistema frágil no presenta capacidad de detectar la presencia de errores, responder a estos, ni tomar acciones correctivas que le permitan recuperarse.
- *Resistente:* Aquel sistema que, ante la presencia de un error o falla, previamente conocido o no, presenta la capacidad de continuar funcionando parcialmente. Tal nivel de madurez no posee como requisito la detección de errores, sin embargo, puede poseerla.
- *Resiliente*: Presenta como requerimiento la capacidad de detectar la presencia de errores - previamente conocidos o no -, tomar acciones apropiadas, y no solo de continuar funcionando parcialmente como un sistema resistente, sino además de ser capaz de regresar a un estado aceptable.
- *Antifrágil:* Un sistema antifrágil no solo es capaz de volver a un estado original previamente definido como aceptable, sino de adquirir además un conocimiento nuevo y evolucionar a partir de este. Los errores que le hacen evolucionar son previamente desconocidos, y producto de que obtiene una ganancia ante la presencia de tales fallas, constantemente se encuentra ante la búsqueda y presencia de nuevas formas de conseguir tales ganancias.

Además, en el contexto del presente trabajo al referirse a un sistema de software no solo se incluyen los ejecutables, sino también configuraciones, entornos e infraestructura de estos, y sobre todo el factor humano, es decir, los equipos encargados de diseñar, implementar, mantener y operar tales sistemas de software.

III. Sistema por estudiar

Características del Sistema

El sistema de software por estudiar se seleccionó tomando como base los siguientes requerimientos:

- Construido con base en el estilo arquitectónico de microservicios.
- Conocimiento de los detalles de arquitectura e implementación tecnológica de este (tecnologías utilizadas, patrones de diseño, decisiones arquitectónicas, protocolos de comunicación, entre otros).
- Conocimiento previo de los procesos de ingeniería de software de los departamentos dueños de este a nivel organizacional. Se define como departamento al conjunto de personal encargado de la implementación, mantenimiento y operación de dicho sistema, los cuales pueden estar distribuidos en diferentes equipos o en el mismo.
- Histórico de experiencias pasadas relacionadas a incidentes y errores (*postmortems*).

Para la descripción del sistema, se decide utilizar la notación C4, dando especial énfasis en los dos niveles iniciales: Diagramas de contexto (Relaciones entre la aplicación, clientes y dependencias terciarias), diagramas de contenedores (distribución de responsabilidades entre cada microservicio, tecnologías utilizadas, y la manera en la que se comunican entre sí), y de menor manera diagramas de componentes (cada sub-parte que compone un microservicio, como interfaces, acceso a datos, caché, etc.), los diagramas de código no se consideran importantes a ser incluidos, debido a que definen comportamientos a nivel de estructura de programación que solamente agregan complejidad en lugar de esclarecer el contexto y comportamiento del sistema de software y sus componentes.

Selección del sistema

Siguiendo los requerimientos anteriormente citados, el sistema de software seleccionado es una serie de microservicios que en conjunto ofrecen un servicio de Web API para sistemas de tipo bursátil, los cuales son diseñados, construidos, implementados y operados por el mismo equipo de ingeniería.

En la figura 1 se muestra el contexto del sistema bajo el estándar C4. La parte central del sistema de software es un REST API, que expone una serie de *endpoints* de estilo *REST* que

permiten acceder a funcionalidades claves, *Clients*, como aquellos terceros externos al sistema de microservicios que consumen tal REST API (por ejemplo, aplicaciones de tipo web o móviles). *Trading Systems* como componentes encargados de realizar acciones relacionadas a órdenes (compra y venta, históricos, estimados). *Data systems* como fuente de datos relacionados a precios, información bursátil y tareas analíticas y, finalmente, *Next Gen Trading Systems* como un conjunto de servicios modernos relacionados a componentes bursátiles avanzados (órdenes condicionales, analítica generativa, *blockchain*, entre otros).

De manera semejante, en la figura 2 se muestra una vista del diagrama de contenedor del sistema, el cual incluye los subcomponentes y microservicios que forman parte de los componentes anteriormente descritos.

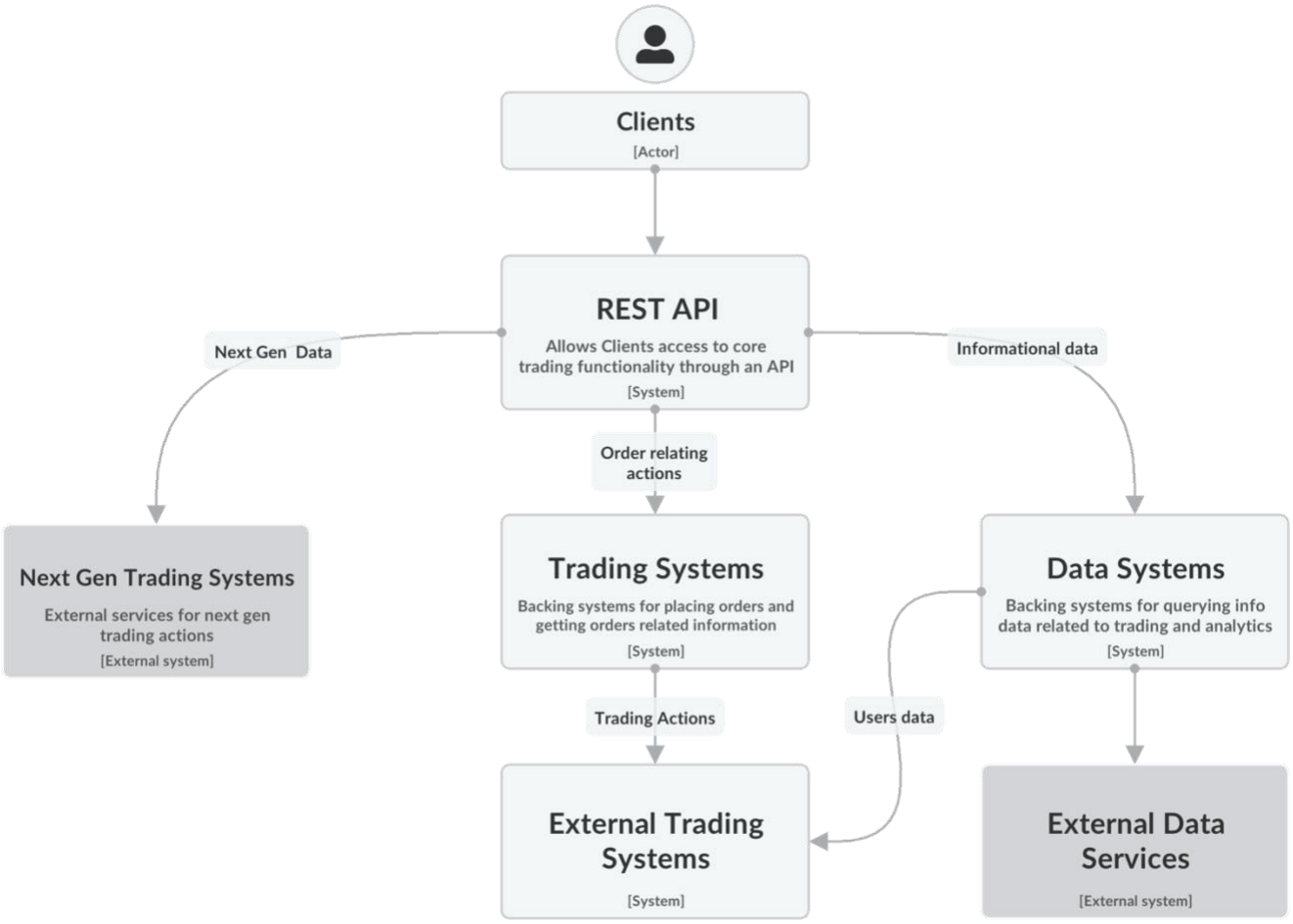

Figura 1: Diagrama de contexto del sistema

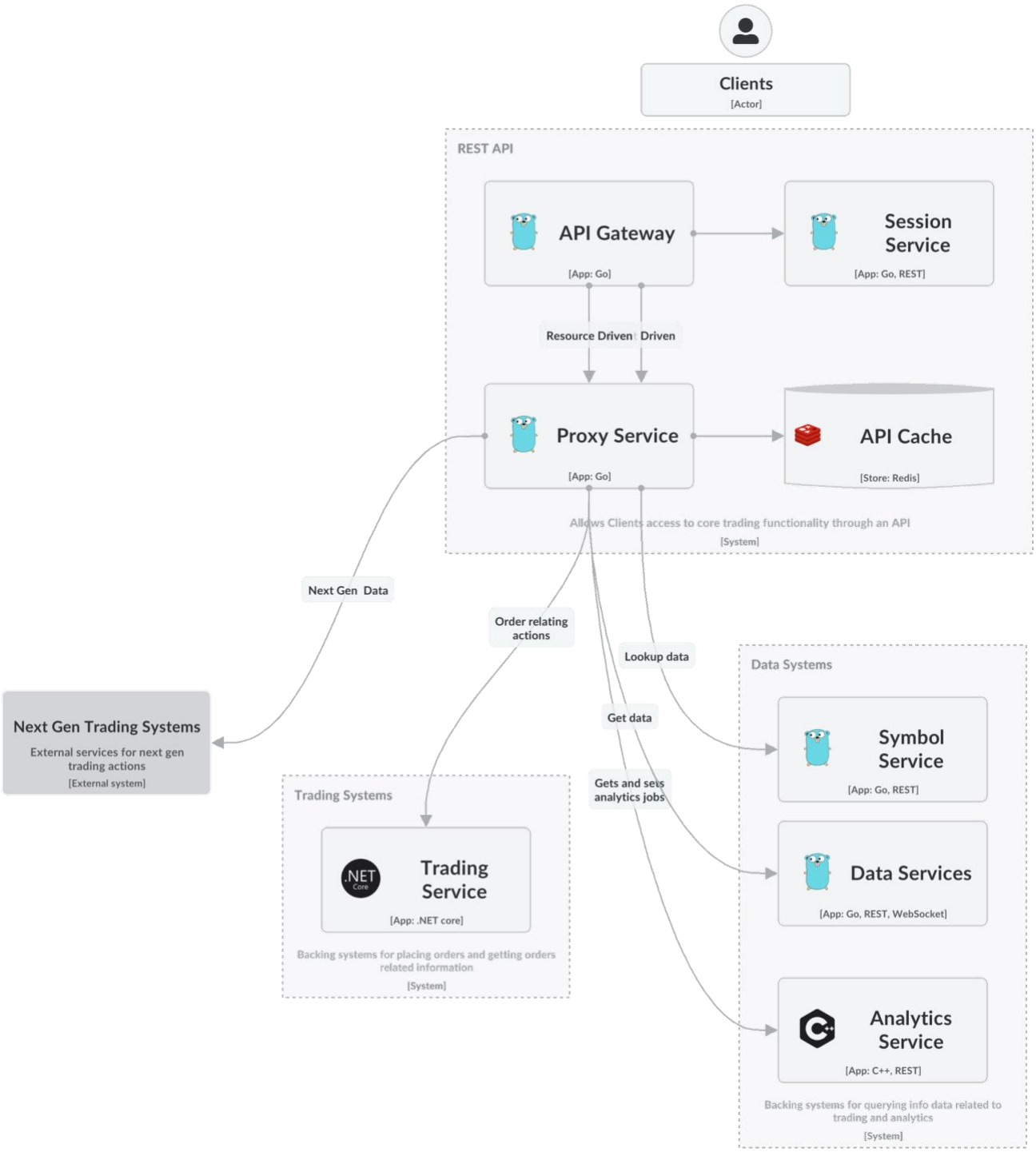

Figura 2: Diagrama de contenedor del sistema

IV. Principios de antifragilidad en software

Monperrus describe el primer conjunto de principios como aquellos relacionados a la presencia de errores en el software, y cómo un sistema con un nivel de antifragilidad es capaz de aprender de ellos.

a. Fault-tolerance and Antifragility

> *A software system with dynamic, adaptive fault tolerance capabilities is antifragile: exposed to faults, it continuously improves.* Monperrus [3]

El énfasis del primer principio consiste en que, para lograr conseguir antifragilidad, nuestro sistema de software primero debe de poseer tolerancia a fallas, y el nivel de madurez es alcanzado cuando tal tolerancia es adaptativa: sin importar la falla a la que sea expuesto, se puede adaptar, funcionar, recuperarse, y terminar en un mejor estado que antes de fallar. Con el fin de alcanzar este punto, se recomienda la adopción de los patrones de diseño a nivel arquitectónico, como los expuestos a continuación, con el fin de alcanzar una base de resiliencia que le permita enfrentarse a errores.

Recomendación: Patrón retry

Debido a la complejidad de una arquitectura de microservicios y a la distribución de componentes en diferentes piezas de infraestructura, así como la dependencia entre estas para comunicarse mediante llamadas sobre la red, es común observar fallas producidas por respuestas inválidas, e inclusive por la falta de respuesta, de un servicio a otro. Algunas de estas fallas son producto de comportamientos internos de los otros componentes, o latencia suficientemente significativa en la red para producir un *timeout*. Sin embargo, no todos los microservicios son capaces de detectar tales respuestas incorrectas o reaccionar a estas de una manera eficiente. Por ejemplo, un microservicio A puede continuar realizando llamadas a un microservicio B aun cuando este nunca responde, generando tiempos de respuesta más altos de los esperados por los clientes.

Por esta razones, la primera recomendación encaminada a alcanzar tolerancia a fallas es utilizar el patrón *retry*, el cual consiste en que un microservicio A realice reintentos de llamadas a

sus dependencias cuando estas fallan por razones previamente definidas (por ejemplo, por un *timeout* o por un error de la familia 5xx[1]). Sin embargo, con el fin de evitar realizar más llamadas de las que son necesarias y no sobrecargar a su dependencia, el microservicio A realiza los reintentos esperando una cantidad de tiempo que crece con subsiguientes llamadas, hasta llegar a un límite predefinido. Tal proceso se ejemplifica con el pseudocódigo:

*reintento = min(limite_tiempo_reintento, tiempo_reintento * (2 * cantidad_de_reintentos))*

Es importante recalcar que un microservicio puede ser por su parte dependencia de múltiples otros microservicios, por lo que es probable que cuando se encuentre en un estado de fallo reciba múltiples reintentos de operaciones por diferentes microservicios que lo consumen, ocasionando picos de llamadas en intervalos de tiempo. Se recomienda no solo agregar un tiempo de espera adicional entre reintentos, sino incluir un tiempo adicional aleatorio entre llamadas. A manera de ejemplo, se recomienda realizar los reintentos siguiendo el pseudocódigo:

*reintento = random(0, min(limite_tiempo_reintento, tiempo_reintento * (2 * cantidad_de_reintentos))*

Con el fin de proveer modularidad y capacidad de reacción adecuada, los límites de tiempo de reintento, así como la cantidad de llamadas que se envían a la dependencia son valores configurables.

En la figura 3 se observa la aplicación del patrón de diseño y la comunicación entre los microservicios *Proxy Service* y *Data Service*, los cuales pertenecen a los componentes *REST API* y *Data Systems* respectivamente.

[1] Los errores HTTP con código de la familia 5xx son considerados por estándar como aquellos en los que el servidor está consciente que ha fallado o no puede realizar la consulta. [10]

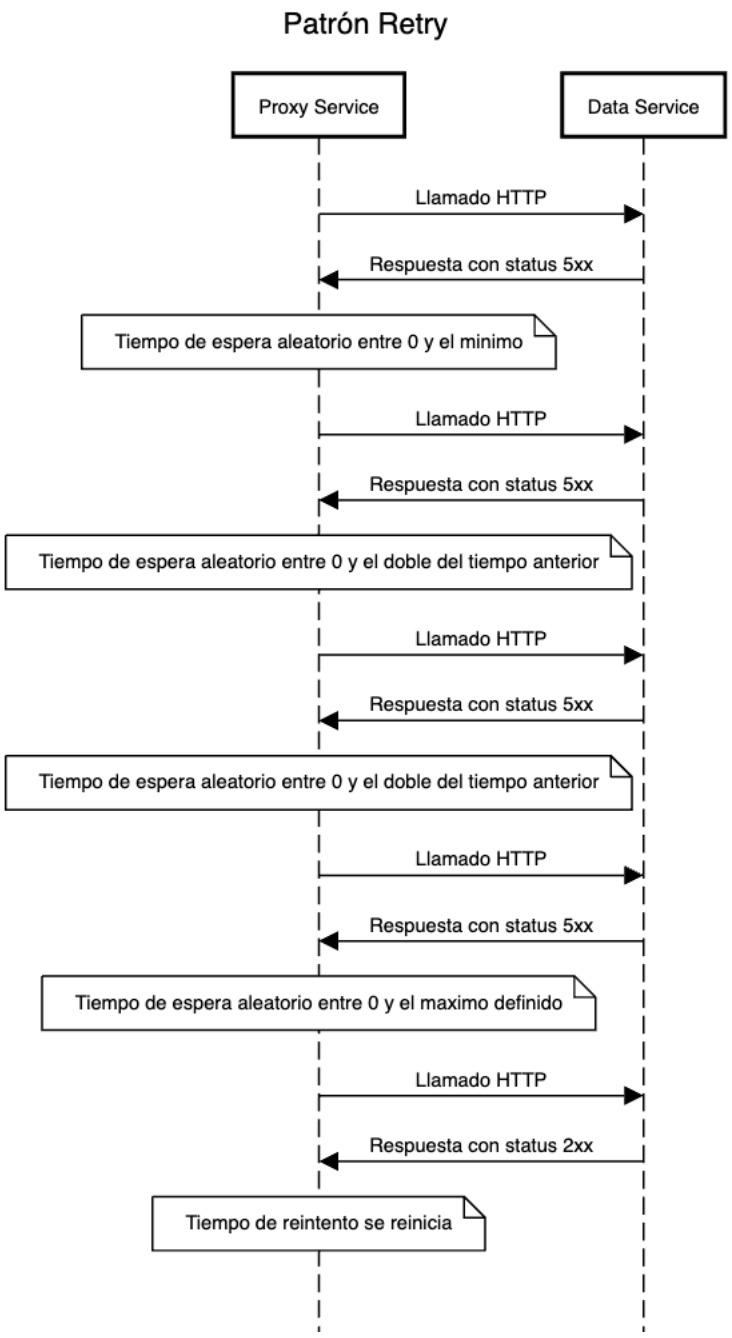

Figura 3: Patrón Retry

Recomendación: Patrón circuit breaker

Una vez implementado un patrón *retry*, en donde se falla de manera automática algunas peticiones con tal de reducir la carga en la red y otras dependencias, así como disminuir el tiempo de respuesta a clientes, se recomienda continuar con la implementación del patrón *circuit breaker*.

Circuit breaker consiste en monitorear las respuestas de una dependencia, verificando por aquellas que posean errores o sean fallidas. Una vez el análisis detecta una cantidad considerable de errores recibidos, se responde automáticamente las siguientes llamadas recibidas de sus clientes con error, y envía solo algunas peticiones a su dependencia, las cuales se realizan cada vez con tiempos más largos entre una y otra, tal y como sucede en el patrón *retry*. Una vez que se pasa un límite de tiempo de reintento, este no aumenta más y los reintentos son constantes (por ejemplo, 1 de cada 10 peticiones son reintentadas). A diferencia de *retry*, las llamadas a red se evitan y fallan de manera preventiva, reduciendo el flujo de llamadas en la red y el tiempo de respuesta a los clientes.

El patrón *circuit breaker* consta de tres estados:

1. Cerrado
    a. Permite realizar llamadas a un componente, y posee un contador de llamadas fallidas.
    b. Si el contador de llamadas fallidas alcanza un límite predefinido, se mueve a estado abierto.
2. Abierto
    a. Falla todas llamadas al componente.
    b. Presenta dos tipos de contadores: de tiempo o de llamadas.
    c. Si cualquier contador excede un límite predefinido, se mueve a medio-abierto.
3. Medio-Abierto
    a. Permite ejecutar una única llamada al componente, y examina su respuesta.
    b. Si la respuesta falla nuevamente, permanece en estado abierto
    c. Si la respuesta es considerada válida, se mueve a estado cerrado.

La ventaja del patrón *circuit breaker* es que, en un estado abierto, reduce el tiempo de respuesta a los clientes, además de ofrecer la opción de activar una funcionalidad alterna o consumir otra dependencia en lugar de la que se encuentra en estado erróneo (por ejemplo, leer un valor de caché en lugar de otra fuente de datos).

En la figura 4 se observa la aplicación del patrón de diseño y la comunicación entre los microservicios *Proxy Service* y *Data Service*, los cuales pertenecen a los componentes *REST API* y *Data Systems* respectivamente.

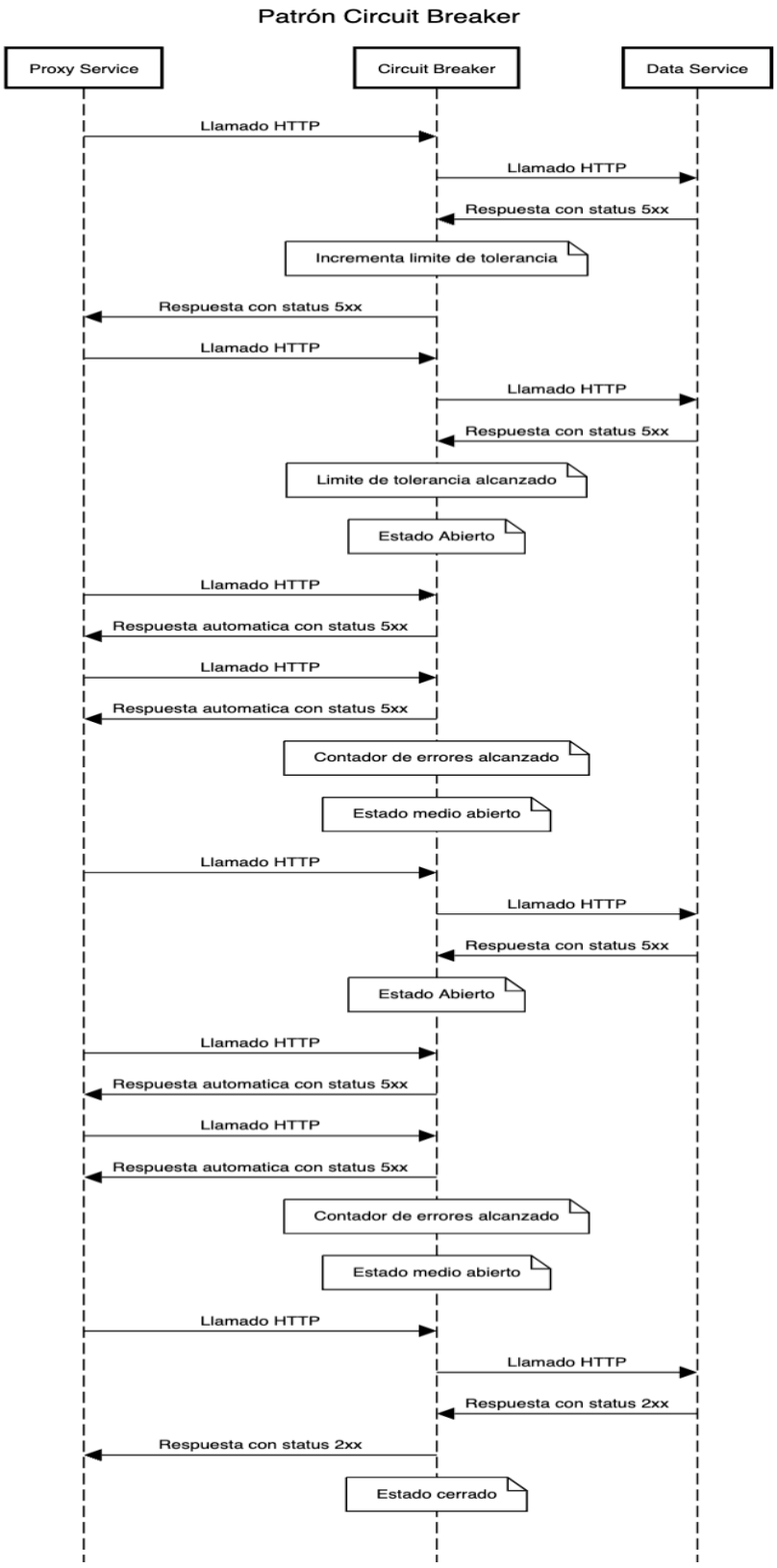

Figura 4: Patrón circuit breaker

b. Automatic Runtime Bug Repair

> *Adaptive runtime repair means "loving errors": A software system with runtime bug repair capabilities loves errors because those errors continuously trigger improvements of the system itself.* Monperrus [3]

La corrección de defectos de manera automática se podría considerar la expresión máxima de la antifragilidad: el software por cuenta propia es capaz de corregirse de manera automática al ser expuesto a una falla, y convertirse en una mejor versión.
Para el análisis de este principio se consideran dos tipos de correcciones: *estado* y *comportamiento*.

Estado

Una corrección o reparación de estado consiste en modificar el estado del programa (registro, *heap*, configuración, *stack*, etc.) durante el tiempo de ejecución [3].

Los componentes del proyecto presentan un nivel básico de reparación automática de errores por estado, son capaces de detectar instancias con problemas de salud, eliminándolas y creando nuevas instancias. Asimismo, algunos componentes presentan detección de configuraciones incorrectas, siendo capaces de aplicar un conjunto de valores predeterminados e informar al equipo en caso de errores provocados por estos. Tales configuraciones incorrectas se evalúan con base en errores de sintaxis, compilación, o una base de conocimiento previa que define cuáles valores son válidos o no.

Sin embargo, ninguno de los componentes presenta un nivel que le permita proactivamente detectar cuándo una configuración genera un error directamente no relacionado a ella (es decir, un error que se da a consecuencia de esta en otro nivel del sistema de software), obtener un aprendizaje y sugerir correcciones. De la misma manera, la detección de estados de salud erróneos en componentes es determinada de manera estática por los integrantes del equipo, y no existe un aprendizaje para conocer o predecir si un nuevo estado no es saludable, más allá de aquellos que han sido construidos previamente.

Como parte de la búsqueda de la antifragilidad, se plantean las siguientes recomendaciones:

Recomendación: Estados de Salud Históricos

Un patrón de diseño común en el diseño de microservicios es el uso de estados de salud, en donde se posee un *endpoint* (usualmente */health* con un código de respuesta HTTP 200) que al consultarse procede a responder con un valor estático informando que el microservicio está en ejecución (adicionalmente se pueden agregar valores dinámicos como versión del servicio, plataforma de ejecución, tiempo de respuesta, etc.). Si bien un estado de salud permite conocer si el servicio se encuentra en ejecución, no necesariamente logra informar si se encuentra funcionalmente saludable. Por ejemplo, si una dependencia requerida por el microservicio se encuentra en estado de error, el *endpoint* no necesariamente informa sobre tal estado de error.

Actualmente existen tres maneras de construir *endpoints* de estado de salud [4]:

1. *Liveness Check:* Envían una respuesta únicamente informando que el microservicio se encuentra disponible.
2. *Shallow Health Check:* Informan que el microservicio tiene altas posibilidades de funcionar. Evalúa recursos como CPU, memoria, almacenamiento, o la presencia de dependencias locales (procesos de monitoreo, bases de datos locales, etc.).
3. *Deep Health Check:* Informan del estado del servicio, sus dependencias locales, y su capacidad de alcanzar otras dependencias o componentes remotos.

Los tres tipos de *endpoints* de estado de salud están presentes en el sistema por estudiar, ofreciendo la detección de errores como tal y en algunos casos se extiende como base para brindar auto-recuperación. Por ejemplo, si el *liveness check* de una instancia falla después de *N* cantidad de intentos, la instancia se recicla y se crea una nueva, por lo que se encuentra en un nivel de resiliente.

Sin embargo, con el fin de alcanzar un nivel de antifragilidad basándose en el principio de *reparación automática de errores por estado*, se propone incluir un factor de aprendizaje que permita definir de manera dinámica los estados de salud, y responder con base en estos. Tales estados se basan en:

1. Exportación automática del estado de una instancia de un componente cuando este es detectado (de manera manual o automática) como no saludable, entre tal información se encuentra:
    a. Versión del aplicativo.
    b. Valores internos de configuraciones.

c. Valores del ambiente de ejecución: versión, registro, configuraciones, servicios en ejecución.
d. Registro de memoria.
e. Logs y métricas generadas en su tiempo de vida.
f. Cantidad de *request* y *response* servidos.
g. Tipo de respuestas recibidas de las dependencias; es decir, si fueron de error o no.

2. Recolección automática de tal información, agregación y almacenaje.
3. Generación de reportes que permitan determinar patrones
4. Generación de nuevas definiciones del estado de salud, que es consumido por las nuevas versiones del microservicio y considerado como parámetro para determinar si se encuentra saludable o no.

Con esta recomendación, un microservicio expuesto a errores o condiciones de falla puede generar un aprendizaje de qué comportamiento realiza ante tal calamidad, y responder desde su estado de salud si se encuentra en error con el fin de proceder a repararlo, por ejemplo, mediante un reciclaje de la instancia de error o una detección por un sistema de monitoreo.

Recomendación: Histórico de configuraciones en relación con comportamientos

El sistema de software por estudiar presenta múltiples configuraciones en todos sus componentes y microservicios con el fin de mantener compatibilidad con los principios de *config* y *backing services* propuestos por el *Twelve-Factor App*[2]*.* Sin embargo, debido a su alta complejidad y gran número de componentes, un cambio de configuración en un solo microservicio puede conllevar efectos cascada en otras piezas de manera directa o indirecta, que no siempre son conocidas u obvias para el operador.

La segunda recomendación se basa en construir un histórico que permita recopilar la asociación de configuraciones, versiones de software y comportamientos del sistema de software como un todo. La propuesta incluye:

1. Determinar la versión específica de cada componente del sistema en ejecución.

[2] *Twelve-Factor App* es una metodología basada en doce principios, propuesta para construir *software-as-a-service* fácil de operar y mantener en condiciones *cloud native*. https://12factor.net/

2. Determinar las configuraciones activas en cada componente, esto incluye desde URLs, *timeouts*, *feature flags*, asignación de recursos, entre otros.
3. Asociar el comportamiento del sistema como un todo ante tales configuraciones.
4. Realizar cambios de configuración de manera controlada, y determinar si el sistema se comporta de la manera aceptada. Por ejemplo, si al deshabilitar un *feature flag* el sistema presenta un correcto manejo del *feature* que ya no existe, u otras dependencias se comportan de una manera no esperada.
5. Realizar un análisis de tales cambios de configuración controlados, así como de aquellos que se producen de manera natural en la continua operación del software.
6. Utilizar los patrones obtenidos del análisis para determinar los posibles impactos de un cambio, así como establecer de manera preventiva qué tipos de cambios generan fallas y cuáles acciones tomar para volver a un estado válido.
7. Poseer una base de configuración que se considere aceptable para la versión en ejecución del sistema, a la cual se pueda volver de manera segura e indolora cuando exista un problema.

De esta manera, se evoluciona de un nivel *resistente* (ante la presencia de una configuración errónea o con efectos inesperados el sistema logra continuar funcionando parcialmente o con capacidad reducida), hacia uno *resiliente* (el sistema es capaz de volver a un estado aceptable de combinación de configuración y versión ante un error o comportamiento indeseado), así como se abre la capacidad de *antifragilidad* (el sistema logra catalogar cuáles combinaciones pueden generar errores, y es capaz de aprender de ellas con el fin de conocer tendencias a errores y maneras de repararse).

Comportamiento

La reparación de comportamiento consiste en modificar la manera en la que una pieza de software se comporta mediante un parche, el cual es generado y aplicado durante el tiempo de ejecución, sin la necesidad de interacción humana [3].

Si bien existe literatura e investigaciones recientes relacionadas a la temática de *self-repair* o *automatic behaviour bug repair* [5], la complejidad existente en la implementación de tales proyectos se considera fuera del tiempo y la cobertura de la presente investigación. En la sección

Limitantes y trabajo futuro se comentan posibles áreas en donde existen avances significativos que podrían ser estudiados más a fondo para futuras recomendaciones.

c. *Failure Injection in Production*

> *A software system using fault self-injection in production is antifragile, it decreases the risk of missing, or rotting error-handling code by continuously exercising it.* Monperrus [3]

Para que un sistema sea considerado antifrágil debe ser capaz de obtener algún aprendizaje y evolucionar al ser expuesto a fallas y errores. Por ello, entre las mejores intenciones para lograr la antifragilidad es exponer constantemente el sistema a errores con el fin de aprender y mejorarlo continuamente. Este principio se basa en provocar las fallas en ambientes reales de producción, con el fin de obtener una vista real de cómo se comporta ante tales incidencias, así como generar una cultura de responsabilidad en los equipos encargados de este, pues se conoce que el sistema se verá expuesto a errores eventualmente.

Las recomendaciones para implementar en el sistema bajo estudio se basan en procesos y técnicas utilizados actualmente en la industria, que buscan tratar los errores como un factor natural en los sistemas de software, y que, en lugar de asumir la ausencia de defectos se basan en una búsqueda permanente de estos, de manera tanto empírica como basada en evidencia o experiencias previas.

Recomendación: Ejercicios de Firedrills

La primera recomendación se basa en una mejora del proceso de OnCall[3] que utiliza el equipo de ingeniería. El proceso actual utiliza un análisis de los *postmortems* de los incidentes, donde la persona encargada del *OnCall* durante el momento del incidente tiene como responsabilidad completar, y expone el contexto, síntomas y pasos por tomar para solucionar el incidente, así como posibles mejoras o acciones por tomar. Tal integrante del equipo realiza una exposición técnica a los demás integrantes, mostrando paso a paso el proceso que realizó para

[3] En un proceso de *OnCall*, una persona o un grupo de trabajo, se encuentra(n) disponible(s) 24/7 de manera rotativa para atender cualquier incidente, tanto detectado de manera automática mediante componentes de monitoreo y alertas, o mediante procesos de escalamiento. Los encargados del *OnCall* tienen la responsabilidad de solucionar el incidente o realizar el escalamiento necesario para que este sea resuelto por un tercero.

alcanzar una solución. Seguidamente, se realiza una reflexión y análisis junto a los demás miembros del equipo sobre posibles mejoras.

La recomendación propuesta aquí consiste en realizar ejercicios de *firedrills* o simulacros de incidentes. En tales ejercicios, se emplea un escenario ficticio de un incidente y se simula el proceso de detectar la causa del problema, llegar a una solución, implementar y validar que el funcionamiento ahora sea adecuado. Para tales ejercicios se manejan los siguientes pasos:

1. Se selecciona un coordinador del ejercicio, quien propone un caso (hipotético o histórico) sobre un posible comportamiento o síntoma de error del sistema de software.
2. Se asigna otro miembro del equipo como “solucionador”, el cual tiene como responsabilidad intentar solucionar el problema, exponiendo los procesos y pasos que realiza.
3. El solucionador explica cómo obtiene la información relacionada a los síntomas presentados, así como el proceso de asociarlos a componentes específicos que podrían estar afectados.
4. Seguidamente, propone una solución o pasos que tomaría para corregir el problema con base en: *SOPs (Standard operation procedures)* documentados previamente, conocimiento propio de la persona basado en experiencias anteriores o funcionamiento interno de los componentes, información esparcida en canales no formales (por ejemplo, correos electrónicos, chats públicos, chats privados, notas).
    a. En caso de identificar que la fuente del error es un componente de un tercero, realiza una explicación de cómo sería el proceso de escalamiento o de contacto a los encargados de tal componente.
    b. Por el contrario, si la persona desconoce cómo solucionar el problema, explica el proceso de escalamiento a otro miembro del equipo.
5. Una vez tomadas las acciones ficticias, se expone cómo se logra determinar si el problema fue solucionado.
6. Finalmente, el coordinador propone y modera una discusión sobre las acciones tomadas, si se puede determinar su eficacia, así como cualquier posible agujero u oportunidad de mejora en los procesos existentes, o si los miembros del equipo poseen las herramientas necesarias para solucionar problemas.

Como ejemplo se toma el siguiente caso:

1. El coordinador presenta un ejercicio ficticio, basado en experiencias anteriores, en donde el equipo es contactado sobre *timeouts* que recibe un cliente que consume un *endpoint* X. Para tal ejercicio se utilizan *logs* y métricas de la última ocasión en que sucedió el problema.
2. El solucionador analiza el *endpoint* que presenta el síntoma de excesivos *timeouts*, y por conocimiento previo de la arquitectura propone analizar el *API Gateway*.
3. Seguidamente, revisa los *dashboards* del componente de interés, donde observa que una gran cantidad de respuestas retornan el código de error 504 *gateway timeout* al llamar al componente *Proxy Service*. Observando los *dashboards* de tal componente encuentra que, si bien se realizan llamadas al *Symbol Service*, la cantidad de respuestas recibidas no concuerdan (se esperaría un promedio de 1:1 *request:response*). Finalmente, en los *dashboards* de *symbol service* no se encuentra ningún indicio o dato útil, sin embargo, en los logs se observan errores de DNS donde no se logra resolver la URL del *data query service*, y una cantidad considerable de errores por *retry.*
4. El solucionador visita la guía de servicio del *Symbol Service*, en donde se encuentran documentados temas como la arquitectura, decisiones de diseño, SLAs, errores esperados, y *SOPs*. Revisando los SOPs, identifica los síntomas del problema y encuentra los pasos para solucionarlo, donde se propone limpiar el *caché* de DNS del *namespace* de *Data Services.*
5. El solucionador propone seguir estos pasos, y como verificación observar los logs del *symbol service* para descartar la presencia de más errores del mismo tipo, así como también los dashboards del *proxy service* para confirmar que se reciben respuestas en un promedio esperado del *symbol service*. Finalmente, una vez que observe que el comportamiento se encuentra en niveles aceptables, propone confirmar con el equipo encargado del cliente que recibía errores que el comportamiento del servicio ahora es aceptable.
6. Como parte de la discusión moderada por el coordinador, el equipo analiza el por qué los sistemas de alerta existentes no son suficientes, y por qué no se toman acciones hasta que un tercero presenta quejas sobre el comportamiento. Se analiza y se propone agregar alertas que detecten una gran cantidad de errores con código de la familia 5xx, en todos los servicios existentes. Asimismo, se propone agregar

un *Kubernetes Operator* a nivel de código en *symbol service* donde este, al detectar un alto número de errores por DNS, limpie automáticamente el *caché.*

En el ejemplo anterior se observa cómo el sistema posee características de un nivel de madurez resistente (ante la presencia de un error algunas funcionalidades de un componente se encuentran deshabilitadas), al ser expuesto el proceso de solucionar un incidente se verifica que el sistema es resiliente (es capaz de volver a su estado original en caso de un error) y se identifican posibles mejoras en este ámbito (tal capacidad ahora se maneja de manera automática). Sin embargo, estas posibles propuestas de mejoras obtenidas al discutir el problema en un ambiente seguro y no urgente (es decir, sin un estado de incidente activo en progreso), permiten obtener un conocimiento que mejoraría sustancialmente el sistema, por lo que se reconoce el proceso de simulacros como una opción para alcanzar un mayor nivel de madurez en antifragilidad (el sistema al verse expuesto a un error - real o ficticio - obtiene conocimiento y es capaz de mejorar).

Recomendación: Implementación de experimentos manuales de Ingeniería del Caos

El proceso de *firedrill* presenta ciertas similitudes a los procesos de Ingeniería del Caos [13]: se fundamentan en la formulación de hipótesis del comportamiento de componentes y los integrantes del equipo encargado del sistema de software, ante síntomas o comportamientos erróneos. Asimismo, se toman eventos y experiencias pasadas como invitación para experimentar. Los experimentos realizados evalúan la capacidad del equipo de solucionar problemas reales en ambientes de estrés, así como el análisis de los procesos y herramientas involucrados para dicho fin. En contraste, los experimentos de Ingeniería del Caos se sustentan en generar hipótesis de cómo se comporta el sistema de software desde una perspectiva técnica ante la presencia de estados de error o de fallas, con el fin de descubrir posible deuda oscura [14] o eventos inesperados.

Debido a que actualmente no se implementan procesos de ingeniería del caos, se propone realizar una serie de experimentos de manera manual, con el fin de respetar el principio *minimize*

blast radius[4], y de generar la suficiente experiencia y confianza en el equipo de manera inicial para que en un futuro se logre evolucionar a experimentos automáticos y constantes.

Los experimentos se proponen de la siguiente manera:

Experimento	Justificación	Preguntas
Desconexión de una instancia	Realizar una desconexión de una instancia presenta los mismos efectos prácticos que otras situaciones como un componente inalcanzable, red de comunicación en estado de error, y fallas parciales de un componente. Además, por experiencias pasadas, es común encontrar durante incidentes algunas instancias o réplicas de un componente en estado de error.	1. ¿Las ejecuciones en progreso permiten la idempotencia? 2. ¿Se puede asegurar la no pérdida de transacciones durante la desconexión? 3. ¿Los *streams* existentes son reabiertos automáticamente en una nueva instancia? 4. ¿Puede el API Gateway efectivamente reasignar el tráfico a una instancia existente? 5. ¿La carga actual en el sistema puede mantenerse sin afectar el rendimiento? 6. ¿Es el sistema capaz de recrear la instancia y cuánto tiempo conlleva? 7. ¿En el caso de fallas en múltiples instancias, cuál es el tiempo de recuperación?
Servicio retorna respuestas inválidas	A diferencia de una desconexión de una instancia, en este experimento se simula el caso en el que una instancia es alcanzable, sin embargo, devuelve respuestas incorrectas (esperadas o no) lo	1. ¿Qué alertas existen para conocer sobre la existencia de tales respuestas? 2. ¿Qué proceso existe para determinar si tales respuestas fueron cacheadas?

[4] *Minimize Blast Radius* es uno de los principios de Ingeniería del Caos que propone que es responsabilidad de ingeniería, al crear experimentos, que estos deben encontrarse aislados lo más posible, con el fin de generar la menor disrupción en los usuarios finales. https://principlesofchaos.org/

	que ocasiona comportamientos no deseados en aquellos que dependen de ella.	● ¿Qué tan complicado es el proceso para invalidar tales cachés? ● ¿Se pueden determinar patrones para evitar cachear tales valores? 3. ¿Se puede determinar desde cuándo se produjeron respuestas o acciones utilizando tales valores incorrectos? 4. ¿Se puede determinar la causa de tales respuestas? 5. ¿Se puede determinar un patrón para evitar realizar tales acciones en el futuro?
Fallo completo de un servicio completo o una dependencia	En lugar de una falla parcial de una instancia de un componente, en este experimento se busca determinar las consecuencias de apagar completamente todas las instancias o evitar el acceso a un servicio por completo. Se espera que, además, las preguntas validadas en el caso de desconexión de una sola instancia se cumplan de la manera esperada.	1. ¿Cuál es el tiempo que se tarda en conocer la falla? 2. ¿Se puede determinar cuántas peticiones fueron atendidas durante ese lapso? 3. ¿Los servicios que dependen de tal componente se comportan de una manera aceptable? Es decir, ¿devuelven un error esperado? 4. ¿Cuál es el proceso para enfrentar tal falla? ● ¿Hay redundancia? ● ¿Hay una alternativa a la dependencia?

Cambio de contrato (API) de una dependencia	En el pasado, se ha producido una cantidad considerable de incidentes producto del cambio de contratos de APIs. Si bien no es técnicamente factible pretender ser capaces de mantener el funcionamiento adecuado del sistema, a pesar de cualquier cambio en dependencias, sí se debe de evaluar cómo se reacciona ante estos.	1. ¿Qué sucede en el caso de cambio de código HTTP en la misma familia, en una misma respuesta? ● Por ejemplo, de un 401 a 403. 2. Si en lugar del código, el cambio sucede en el contenido de un *error body*, ¿Cuál es el efecto? 3. Si se reciben valores que no pueden ser analizados y convertidos (cambios de línea o de separadores, *enums*, entre otros), donde antes no era así), ¿Cómo se comporta la aplicación? ● ¿Se muestra un error aceptable? ● ¿Ocurre un error o pánico fatal que conlleva la caída total del componente? ● ¿Sucede un comportamiento que no es el esperado ni deseado? Por ejemplo, valores default.
Diferentes versiones de un componente propio ejecutando en diferentes instancias paralelamente	Durante un incidente pasado, se encontró que una versión del aplicativo en ejecución no era la correcta, es decir, se hizo un *release* de la aplicación sin embargo no se eliminó el *stack* anterior ni se realizaron actualizaciones en los componentes encargados del enrutamiento para redirigir	1. ¿Son las herramientas de pruebas automáticas suficientes para determinar en todos los casos si algunas instancias de un componente poseen código incorrecto o antiguo ejecutándose? 2. ¿Generan las versiones incorrectas suficientes diferencias en comportamientos para ser detectadas?

	las peticiones a la nueva versión. Esta problemática ocasionó que el comportamiento fuese diferente al esperado, sin embargo, de una manera tan sutil que no logró ser detectado correctamente. Este experimento analiza el impacto y capacidad de resiliencia en caso de que sólo algunas instancias de un componente sean diferentes, convirtiendo el problema en uno más difícil de detectar mediante técnicas convencionales.	● ¿Las diferencias de comportamiento poseen un impacto negativo en el uso del sistema de software? 3. ¿Qué tan rápido es el equipo en encontrar tal problema, solucionarlo, y evitar que vuelva a suceder en el futuro? 4. Si en lugar de versiones de código, son configuraciones (*feature flags,* puertos, URLs, variables de entorno, etc) las que son diferentes, ¿aplican las mismas respuestas a las preguntas anteriores?
Cambios de configuración inesperados	Producto de la utilización de prácticas arquitectónicas como *backing services* mencionada anteriormente, los componentes y microservicios definen sus dependencias mediante el uso de valores. La aplicación de estos experimentos busca analizar el impacto de generar cambios en tales dependencias que requieran modificar esas configuraciones.	1. Si una dependencia cambia su URL, ¿Las alertas de errores producidas al intentar acceder a esta son lo suficientemente claras para llegar a la conclusión de cuál fue el origen del problema? 2. ¿Los miembros del equipo tienen acceso oportuno para realizar los cambios de configuración? 3. ¿Los cambios de configuración se pueden replicar entre múltiples instancias de un mismo componente o requieren intervención manual en cada uno de ellos?

		4. ¿Los cambios de configuración generan nuevas versiones manteniendo el principio de *infraestructura inmutable*? 5. ¿Tales cambios de configuración, así como las razones del por qué se realizan y su contexto, son almacenados a futuro?
Pérdida temporal en las capacidades de monitoreo	La agregación de métricas y análisis con el fin de generar alertas es dependiente de sistemas de terceros de los cuales los equipos de ingeniería no son los encargados. Esta propuesta busca analizar el caso en que estos sistemas de terceros sufran caídas o no se encuentren disponibles.	1. ¿Si hay una caída en los canales de alerta regulares, son los equipos capaces de conocer si hay un incidente? 2. En el caso de la caída de estos canales, ¿Qué tan rápida y oportunamente pueden los equipos darse cuenta de que no están disponibles? 3. Si hay una pérdida de métricas o *logs*, ¿Son los equipos capaces de realizar depuración en el sistema para encontrar cualquier error?

V. Principios de antifragilidad en los procesos de desarrollo de software

El otro conjunto de principios descritos por Monperrus son aquellos directamente relacionados con el proceso de construcción del sistema de software.

a. *Test-Driven Development*

> *When a bug is found, a test that reproduces the bug is first written; then the bug is fixed. The resulting strength of the test suite gives developers much confidence in the ability of their code to resist changes.* Monperrus [3]

Test-Driven Development es uno de los procesos de software más estudiados y propuestos como una posible alternativa a las metodologías tradicionales, principalmente por la confianza que genera al basarse en la construcción temprana de casos de pruebas y un uso de herramientas de automatización que permite la constante verificación y confianza a la adhesión de requerimientos tanto funcionales como no funcionales. Monperrus toma esta metodología como base para proponer un principio, en el que, al construir pruebas de manera extensiva, permitimos exponer al software a cambios constantes teniendo un nivel de confianza adecuado donde la probabilidad de introducir nuevos errores será baja.

Las propuestas sobre este principio buscan no solo incrementar el nivel de confianza que se tiene mediante la construcción preventiva de pruebas, sino que, además, busca elevar los procesos de pruebas a un nivel mayor de antifragilidad donde ellos mismos mejoren al ser expuestos a errores.

Recomendación: Service-Level Fault Injection

La primera recomendación se basa en la utilización de una herramienta llamada *Filibuster*[5] expuesta en el artículo *Service-Level Fault Injection Testing* [6]. Esta herramienta consiste en una versión modificada - o *fork* - de las bibliotecas (“librerías”) de OpenTelemetry[6] que, en lugar de realizar *tracing* de servicios y dependencias terceras a un componente, inyecta fallas como errores inesperados, latencia, desconexiones y demás, con el fin de evaluar el comportamiento del servicio ante la presencia de un error en sus dependencias.

El proceso para implementar la recomendación es iniciar con la instalación del *fork* en un único componente, en un ambiente de pruebas destinado únicamente con este fin. Seguidamente

[5] El Proyecto Filibuster se puede encontrar en https://github.com/filibuster-testing

[6] OpenTelemetry es un proyecto *Open Source* desarrollado por la *Cloud Native Computing Foundation*, que consiste de un conjunto de herramientas que permiten el monitoreo (y un mayor nivel de observabilidad) de sistemas distribuidos utilizando técnicas como métricas, *logs* y *tracings*.

se ejecuta una batería de pruebas automáticas preexistente, con la modificación de que se validan tanto por respuestas correctas, como por errores esperados.

Si se producen errores o comportamientos no aceptables producto de que se obtuvieron respuestas incorrectas al llamar a las dependencias del componente a evaluar, la batería de pruebas automáticas los captura y da a entender que existen comportamientos no esperados que no necesariamente se reconocen. A partir de ahí, se realiza una correlación entre las llamadas con respuestas incorrectas y los errores inyectados por la herramienta *filibuster*, utilizando identificadores ya existentes en la biblioteca ("librería") estándar de OpenTelemetry como *tracingid*.

El *Service-Level Fault Injection testing* se puede eventualmente integrar como un proceso del CI/CD, obteniendo un mecanismo de inyección de fallas automatizado similar a la ingeniería del caos, con menores costos y riesgos de implementación.

A diferencia de procesos y herramientas ya en práctica, como pruebas unitarias o de integración, esta propuesta utiliza una estrategia de exploración y búsqueda constante de nuevos escenarios de error, los cuales permiten obtener como resultado un sistema de software mejorado. Por ello, se considera que la aplicación de esta propuesta acerca a un mayor nivel de madurez de antifragilidad.

Recomendación: Pruebas de Rendimiento orientadas a fallas

Actualmente, el proceso de pruebas de rendimiento se utiliza para determinar si el sistema o algunos de sus componentes son capaces de soportar una carga determinada con un rendimiento aceptable cumpliendo una serie de objetivos de SLAs[7] definidos a nivel corporativo.

Esas pruebas no tienen como objetivo determinar el nivel máximo de estrés que un componente puede soportar antes de entrar a un estado de falla, ni cómo se comporta el sistema ante tales niveles. La idea de utilizar pruebas de rendimiento es obtener un conocimiento de hasta qué punto el sistema se puede enfrentar en cuanto a cargas de trabajo, y cómo el equipo está preparado para atender una demanda mayor que la normal. La recomendación se basa en utilizar pruebas de rendimiento con el fin de conocer puntos tales como:

[7] SLA = Service Level Agreement, que se traduce como Acuerdo de Nivel(es) de Servicio. Es un contrato que establece la cobertura de un conjunto de servicios.

1. El comportamiento de cada componente ante niveles altos de estrés, no si cumplen o no los SLAs, sino cuáles características o comportamientos presentan cuando se llevan a niveles iguales o mayores al peor caso esperado.
2. El comportamiento del sistema como un todo ante niveles altos de carga en algunos de sus componentes, así como sus dependencias.
3. Confirmar si tales comportamientos son aceptables. Por ejemplo:
    - La latencia y tiempo de respuesta no se encuentra en un nivel aceptable, pero se devuelve un código de error esperado.
    - Producto del gran uso de recursos, se crean condiciones de carrera que terminan en comportamientos o errores inesperados.
    - El alto uso de memoria/CPU hace que algunas transacciones fallen, entonces:
        - La transacción se ejecuta, pero el cliente recibe un error (o falta de respuesta) que da a entender lo contrario.
        - La transacción no se ejecuta y el cliente recibe un error relacionado.
        - La transacción no se ejecuta y el cliente recibe un error no esperado.
4. Determinar si el equipo operador de tal sistema es capaz de enterarse a tiempo sobre una carga, y si las acciones a tomar son aceptables.

b. Bus Factor

> *If a key developer is hit by a bus (or anything similar in effect), could it bring the whole project down?*

El factor humano en el desarrollo de software es un componente frágil, es propenso a errores y, a menos que exista intención, rara vez se mejora a partir de estos. El principio de *bus factor* expone una realidad de muchos equipos: si un miembro clave ya no forma parte de un proyecto, es posible que aparezcan diversas problemáticas producto de conocimiento o experiencia exclusivos de tal integrante. Las recomendaciones con base en este principio buscan identificar tales conocimientos, cómo el equipo es capaz de afrontar la ausencia de acceso a este, y mejorar el proceso de construcción y operación del software.

Recomendación: Firedrills sin miembros del equipo

Similar a la recomendación de ejecutar *firedrils* o simulacros de incidentes con el fin de conocer qué tan preparado se encuentra el equipo ante posibles incidentes reales, así como encontrar agujeros en los procesos y tener discusiones técnicas de cómo mejorar el producto, aquí se propone la recomendación de realizar tales ejercicios sin involucrar a miembros clave del equipo. Por ejemplo, aquellos con mayor experiencia o *seniority* como líderes técnicos, o aquellos que se consideren *SME (subject matter experts*, expertos en la materia*)* sobre componentes, microservicios o funcionalidades.

Asimismo, se busca que el incidente utilizado para el simulacro se base específicamente en conocimiento o áreas donde el integrante *solucionador* del incidente posea la menor experiencia posible.

Recomendación: Movimientos internos de personal

El *bus factor* es ocasionado porque conocimiento y experiencia claves se concentran en un subconjunto de miembros de un equipo u organización. Por ello, se propone identificar y reducir ese factor de riesgo en un sistema de software, al enfrentarlo a errores producidos por dicho factor, así: rotar el personal de ingeniería de tal manera que se evidencie la pérdida de conocimiento en el equipo. Tal recomendación no suele ser considerada práctica por aspectos humanos como la pérdida de moral en un equipo, menores oportunidades de desarrollo profesional (*career path*) a mediano o largo plazos, reducción de productividad, y descontento con el cambio constante de ambiente.

En contraste, una alternativa a la recomendación anterior es realizar movimientos internos en un mismo equipo. Es normal observar que ciertos integrantes por distintas razones obtienen conocimiento específico sobre reglas de negocio u operaciones técnicas sobre componentes y microservicios concretos y, por afinidad, constantemente se les asigne trabajo relacionado con tales componentes. En lugar de mantener *SME* sobre áreas específicas, un miembro de un equipo no trabajará más de dos tareas consecutivas relacionadas a un componente.

Al rotar el personal de manera interna, se produce una constante pérdida de contexto e información sobre componentes o funcionalidades, lo que obliga al integrante que se encuentra ahora encargado de la nueva tarea a buscar aquel conocimiento que no posea y necesite en las múltiples fuentes de datos y conocimiento existente. En caso de no encontrarse previamente en un repositorio, pero sí en el conocimiento de algún otro integrante, se procede a documentar.

Con esta recomendación, en un principio será observable la presencia de errores en el proceso de desarrollo producto de información y contexto existentes en islas de información. Sin embargo, con el pasar del tiempo las bases de conocimiento documentadas irán en aumento - no solo de tamaño sino de calidad -, obteniendo un equipo con un *bus factor* menor. Esto en contraste a la situación anterior a la implementación de esta recomendación, en donde no solo existe un *bus factor* mayor, sino que además se desconoce cuánto y cuál es el conocimiento concentrado en unos pocos integrantes del equipo.

Recomendación: Firedrills de dependencias

Debido a la existencia de dependencias de terceros equipos ajenos a la construcción y operación de los sistemas por estudiar, y a la posibilidad de fallas en tales componentes, se propone la idea de realizar ejercicios de ***firedrills*** en donde el error hipotético por evaluar es una falla de un sistema tercero. Tal ejercicio permite determinar si el equipo de desarrollo primario conoce los pasos para realizar el contacto (dónde, cómo, quién), verificar previamente si hay un error reportado, así como el tiempo de respuesta de los terceros y sus planes a evaluar.

Una posible recomendación adicional es la aplicación de experimentos de ingeniería del caos similares, en donde se desconecten los canales de comunicación de las aplicaciones o mediante un ejercicio conjunto se inyecten errores en las dependencias. Los ejercicios en conjunto se pueden realizar tanto estilo ***white box*** (ambos equipos conocen que se inyectará la falla), como ***black box*** (solo un equipo o algunos miembros seleccionados de ambos conocen sobre el ejercicio).

VI. Discusión

a. Conway's Law (La Ley de Conway)

> *"Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations".*
> Conway [11]

En 1968, Melvin Conway [11] expuso que el diseño de sistemas de software está estrechamente relacionado con la distribución organizacional; es decir, la arquitectura de un sistema de software depende de la forma en la que los equipos de diseño, construcción, mantenimiento y operación están organizados. Es importante notar que no solo el diseño arquitectónico se ve afectado por la distribución organizacional, sino además por los procesos adoptados por tales organizaciones y los equipos de software que la conforman.

Por tal razón, es observable que, con el fin de lograr software antifrágil, es necesario aceptar que la presencia de *procesos* antifrágiles es un requisito fundamental en los equipos de software.

Un equipo antifrágil es capaz de aceptar que la presencia de errores es una constante en la producción de software, por lo que buscan nutrir una cultura donde tales errores no son vistos con repercusiones negativas sino como posibilidades de mejora [7]. La transparencia y responsabilidad sobre los sistemas de software es parte fundamental de la cultura organizacional, al aceptar los errores como oportunidad de mejora se insta a tener transparencia en estos con el fin de obtener nuevo conocimiento, tomar acciones y por ende resultar en un sistema mejor, un principio clave de la antifragilidad.

b. Limitantes y trabajo futuro

Las limitantes actuales en antifragilidad se observan en el principio de *Automatic Runtime Bug Repair*, especialmente en el subconjunto de reparaciones de comportamiento, debido a la complejidad innata de conseguir que un sistema sea capaz de no solo determinar cuál es su estado esperado y cómo presenta una desviación de este, sino además de sintetizar una posible solución, validarla tanto ante la posible desviación como en cuanto a posibles efectos secundarios, y finalmente realizar una aplicación de tal solución, todo esto sin la intervención humana.

El estado del arte actual [5] sugiere el uso de oráculos que permitan tanto detectar la presencia de defectos, fallas o errores, así como que permitan la detección de que no fueron introducidos nuevos errores o defectos durante la reparación automática. Tales oráculos permiten determinar si el resultado obtenido tras una ejecución es correcto, es decir, son validaciones relacionadas directamente con la especificación del software. El uso de oráculos propone técnicas como pruebas automáticas, validación de diseños por contrato [15], y análisis de modelos de software.

Otra técnica propuesta el uso de análisis estático, en el cual el código es analizado antes o durante el tiempo de compilación con el fin de encontrar posibles defectos, son definidos mediante restricciones del lenguaje de programación (por ejemplo, tipado seguro y validación de parámetros), *linters*, pruebas automáticas y unitarias, y bases de conocimiento previamente definidas (por ejemplo, que permitan verificar la presencia de versiones de software y dependencias obsoletas o inseguras). La implementación actual en procesos de CI/CD permite realizar algunas de estas validaciones e inclusive reparaciones con poca necesidad de intervención humana (por ejemplo, actualizando bibliotecas, detectando posibles condiciones de carrera, proponiendo correcciones cuando hay un error de compilación), y de manera regular aquellos desarrolladores activos en el sistema de software obtienen conocimiento sobre tales validaciones y errores producidos, reduciendo su probabilidad de volver a cometerlos en un futuro. Con ello se van mejorando los procesos y presentando características de sistemas antifrágiles. Asimismo, se sabe que existen técnicas reconocidas de automatización de reparaciones de defectos que pueden ser estudiadas e inclusive implementadas a nivel industrial.

Una alternativa prominente en la literatura y con un gran posible valor en sistemas de software basados en microservicios es el uso de oráculos que validen diseños por contratos mediante pre y post condiciones, es decir, una pieza de software como un microservicio ofrece un contrato o interfaz - por ejemplo, mediante un API REST -, y al encontrarse un defecto este es reparado de manera automática con la condición de que la interfaz se siga cumpliendo. Sin embargo, presenta limitaciones como la dificultad de acertar por efectos secundarios no implícitos o visibles en el contrato, o limitaciones del lenguaje en que se especifiquen los contratos (incompletitud o incomputabilidad, por ejemplo).

VII. Conclusiones

Los principios de antifragilidad proponen una base de características necesarias que posea un sistema de software para que no sea solo capaz de resistir impactos por presencia de defectos o errores, sino además resultar en una nueva versión mejor. Tales principios no se refieren únicamente a características propiamente técnicas o de diseño del sistema de software, sino que también incluyen a los procesos y a las organizaciones de ingeniería utilizados en la construcción, mantenimiento y operación del sistema.

Gracias a la investigación realizada, fue posible presentar aquí un conjunto de mejoras bajo cada principio, las cuales se basan en tomar los errores y defectos de un sistema como

posibles puntos de aprendizaje, de los cuales se pueda generar nuevo conocimiento con el fin de mejorar el sistema de software constantemente. Asimismo, se presentan recomendaciones que permiten la búsqueda continua de errores proactivamente, con el propósito tanto de mejorar el software existente y sus procesos relacionados, como de reducir la deuda oscura en el software.

VIII. Referencias

[1] S. Newman. *Building Microservices*. Sebastopol: O'Reilly Media, Inc., 2015.

[2] R. Vitillo. Understanding Distributed Systems, 2nd ed. 2022.

[3] M. Monperrus. «Principles of Antifragile Software». *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming,* pp. 1-4, 2017.

[4] M. A. Titmus. *Cloud Native Go*. Sebastopol: O'Reilly Media, 2021.

[5] M. Monperrus. «Automatic Software Repair: a Bibliography». *ACM Computing Surveys,* vol. 51, nº 1, pp. 1-24, 2017.

[6] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller y R. Padhye. «Service-Level Fault Injection Testing», *SoCC '21: Proceedings of the ACM Symposium on Cloud Computing,* pp. 388-402, 2021.

[7] A. Tseitlin. «The Antifragile Organization: Embracing Failure to Improve Resilience and Maximize Availability». *ACM Queue,* vol. 11, nº 6, pp. 20-26, 2013.

[8] D. Russo y P. Ciancarini. «Towards Antifragile Software Architectures», *Procedia Computer Science,* pp. 929-934, 2017.

[9] D. Russo y P. Ciancarini. «A Proposal for an Antifragile Software Manifesto». *Procedia Computer Science,* nº 83, pp. 982 - 987, 12 Mayo 2016.

[10] Internet Engineering Task Force (IETF). «RFC 9110 - Section 15.6. Server Error 5xx». Junio 2022. [En línea]. https://www.rfc-editor.org/rfc/rfc9110.html#name-server-error-5xx. [Último acceso: 10 Diciembre 2023].

[11] Melvin E. Conway. «How Do Committees Invent?». *Datamation*, vol. 14, n° 5, pp. 28-31, Abril 1968. [En línea]. http://www.melconway.com/Home/Committees_Paper.html

[Copia del original publicado]. http://www.melconway.com/Home/pdf/committees.pdf [Último acceso: 26 de febrero 2024]

[12] Tom Limoncelli. «Resilience Engineering: Learning to Embrace Failure». *ACM Queue*, vol. 10, n° 9, 13 setiembre 2012.

[13] Nora Jones & Casey Rosenthal. *Chaos Engineering*. Sebastopol: O'Reilly Media, 2020.

[14] D. D. Woods. *STELLA Report from the SNAFUcatchers Workshop on Coping With Complexity*. [En línea]. https://snafucatchers.github.io/

[15] Bertrand Meyer. «Applying "Design by Contract"». Computer, vol. 25, n° 10, pp. 40–51. Octubre 1992, IEEE.[En línea]. https://se.inf.ethz.ch/~meyer/publications/computer/contract.pdf