UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

TÉCNICAS PARA OBTENER OBSERVABILIDAD EN UNA ARQUITECTURA BASADA EN MICROSERVICIOS

Samuel García González

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"TÉCNICAS PARA OBTENER OBSERVABILIDAD EN UNA ARQUITECTURA DE MICROSERVICIOS"

Autor: **Samuel García González**

Director: **Antonio Moratilla Ocaña**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

A Patricia, por acompañarme en el final de esta etapa y nuevo comienzo, ¡a volar!

ÍNDICE

1. INTRODUCCIÓN

El panorama actual del desarrollo software está copado por las soluciones en la nube, y la tendencia no parece que vaya a cambiar, a menos a corto plazo. Ver ejemplos de éxito día tras día que alojan sus aplicaciones en estos entornos, junto con arquitecturas fascinantes para hacer frente a la demanda que solicitan sus usuarios es poco menos que llamativo.

La fascinación propia de un ingeniero nos lleva a pensar en los equipos que no sólo desarrollan, sino que además gestionan estas aplicaciones, y nos genera una gran cantidad de dudas. ¿Cómo serán capaces hacer funcionar unos sistemas que den servicio estable a tantos usuarios simultáneos? ¿Cómo consiguen que sus sistemas sean eficientes y no agujeros negros de poder de computación? ¿Cómo pueden saber si su sistema está funcionando correctamente? Todas estas preguntas son las que motivaron este trabajo.

La forma de construir aplicaciones con el nacimiento de la computación en la nube cambió drásticamente y, como posteriormente describiremos, la arquitectura basada en microservicios ha llegado para reinar como podemos ver en esta encuesta de O'Reilly [1] de los que usaban esta arquitectura, el 92% reportaron éxito con su adopción. Esta arquitectura es el corazón de este trabajo, ya que en su esencia es un sistema distribuido y, como tal, conocer el funcionamiento del sistema es muy complicado, más aún si contamos con que los recursos hardware sobre los que se ejecuta no están siquiera presentes, sino que "existen" en la nube.

La observabilidad es un concepto que se ha popularizado recientemente, sobre el que se está dando un debate muy productivo, ya que promete responder a las necesidades de conocer el funcionamiento de su sistema que tienen los desarrolladores en este mundo nativo en la nube y con aplicaciones diseñadas como sistemas distribuidos. En este trabajo analizaremos críticamente este concepto, repasaremos las implementaciones más novedosas del mismo y su posible evolución.

2. OBJETIVOS

El propósito principal de este Trabajo de Fin de Máster es analizar las técnicas más novedosas y populares para obtener observabilidad en una arquitectura basada en microservicios.

Para analizar estas técnicas se realizará un estudio de los fundamentos que existen detrás de ellas, tanto tecnológicos como matemáticos, para conocer cómo las arquitecturas basadas en microservicios pueden beneficiarse del uso de estas.

Una vez conozcamos el trasfondo teórico de las diferentes técnicas, pasaremos a realizar un análisis de las herramientas más populares que implementen estas técnicas, pudiendo ver las diferentes características y aproximaciones de cada una de ellas para conocer los casos de uso para los que estén pensadas y construidas.

Por último, tras conocer en profundidad todos los conceptos respecto a la observabilidad en sistemas distribuidos, se investigará sobre los posibles nuevos caminos y tendencias que seguirá esta técnica, proponiendo líneas de investigación o conceptos recientes y novedosos.

Para establecer una metodología de trabajo y conseguir estos objetivos, establecemos una lista de ellos para concretarlos:

- Describir de la arquitectura basada en microservicios según el estado del arte y buenas prácticas.
- Establecer claramente la diferencia entre sistemas monolíticos y microservicios (sistemas distribuidos), para comprender la necesidad de la observabilidad.
- Conocer los avances y necesidades que llevaron al surgimiento de la observabilidad, analizando las debilidades de los sistemas monolíticos y el desafío de la adopción de los microservicios.
- Definir el concepto y cualidades que caracterizan a la observabilidad.
- Diferenciar la observabilidad de la monitorización, debido al cambio de paradigma que supone.
- Estudiar los fundamentos y conceptos teóricos tras la trazabilidad distribuida, los logs y las métricas como técnicas para ganar observabilidad.
- Realizar un análisis histórico de la trazabilidad distribuida, los logs y las métricas para comprender los avances de las técnicas y poder desarrollar un estudio del arte de estas, además de repasar sus buenas prácticas.
- Indagar sobre las herramientas más populares o con propuestas más novedosas que implementen la trazabilidad distribuida, los logs o las métricas con el objetivo de conseguir observabilidad.
- Desarrollar un análisis crítico relacionando el estado del arte de los microservicios, la observabilidad y las técnicas analizadas para proponer posibles líneas de investigación o desarrollo.
- Investigar sobre las tendencias más novedosas y rompedoras con los conceptos presentados y evaluar sobre su posible evolución y aceptación.

3. ESTADO DEL ARTE

En este apartado trataremos de abordar los conceptos clave de este trabajo, dando una visión actualizada sobre todos ellos y sus aplicaciones más novedosas y con mayor potencial de desarrollo futuro.

3.1 MICROSERVICIOS

Los microservicios son un tipo de arquitectura relativamente reciente y con una popularidad en aumento constante, como representa el mercado de microservicios en la nube, que creció en un 22,5% sólo en 2020 [2]. La tendencia nos indica que los desarrolladores prefieren alejarse de soluciones tradicionales, como lo es alojar localmente sus aplicaciones, y gravitan hacia soluciones en la nube. Esta tendencia lleva a diferentes expertos a afirmar que durante 2022 el 90% de las aplicaciones serán desarrolladas usando esta arquitectura [2].

El término "microservicios" fue discutido por primera vez en un taller de arquitectos de software en mayo de 2011 en Venecia, que venía a describir un nuevo estilo de arquitectura en el que los participantes habían realizado investigaciones y primeras aproximaciones. Este mismo grupo decidió en mayo de 2012 que el término "microservicios" era el más apropiado para representar este estilo. Como primera mención fuera de este grupo tenemos la conferencia de James Lewis en la 33rd Degree de Cracovia, titulada "Micro services - Java, the Unix Way" [3].

El ingeniero de software Robert C. Martin mucho tiempo antes de la adopción del término "microservicios", concretamente en el año 2003, aportó uno de los pilares más importantes para esta arquitectura, el llamado "Principio de Responsabilidad Única". Este principio establece que una clase debe tener una única razón para cambiar [4]. Robert C. Martin aclaró que este principio también está orientado a las personas y sus responsabilidades, dado que, aunque una misma persona haga uso de un sistema, si existen diferentes roles, como por ejemplo el de contabilidad o el administrador de base de datos, cada uno de esos roles debe estar separado en su propio módulo, haciendo efectivo el Principio de Responsabilidad Única [5].

La definición formal más aceptada de los microservicios como arquitectura es, en palabras de James Lewis y Martin Fowler, "El estilo de arquitectura de microservicios es una aproximación para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno ejecutando sus propios procesos y comunicándose con mecanismos ligeros, comúnmente a través de APIs de recursos HTTP. Estos servicios son creados en base a las capacidades del negocio y son independientemente desplegables por un sistema completamente automatizado. Estos servicios poseen una gestión centralizada mínima, los cuales pueden estar escritos en diferentes lenguajes de programación y usar diferentes tecnologías de almacenamiento de datos" [6].

Este principio surgió junto con otros similares, con la finalidad de reducir la complejidad de desarrollo y, sobre todo, el gran acoplamiento al que se enfrentaban los sistemas debido a su crecimiento continuo. Antes de la arquitectura de los microservicios, la arquitectura predominante en el software era la "monolítica". En esta arquitectura, el software era diseñado para ser autocontenido (lo que conllevaba ese gran acoplamiento), con lo cual, todas las piezas o componentes de este eran necesarias para tanto ejecución como compilación. Esto, además de conllevar unos tiempos dilatados para los procesos mencionados, causaba el efecto colateral de que cualquier mínimo cambio en el código

fuente conllevaba una recompilación y redespliegue del sistema al completo [7]. Uno de los principales problemas a los que se enfrentaba la arquitectura monolítica era la escalabilidad que requerían ciertas aplicaciones, cada vez con mayor demanda debido a la popularización de Internet. Escalar verticalmente (dotar al sistema de más recursos hardware) una aplicación monolítica conlleva unos grandes costes económicos y de planificación de recursos. Si optamos por escalar horizontalmente una aplicación monolítica, necesitaremos construir mecanismos adicionales para gestionarla (diferentes réplicas del sistema), como por ejemplo un balanceador de carga, además de los costes asociados. Cualquiera de los escenarios incrementa exponencialmente las fallas intrínsecas a este tipo de arquitecturas, haciendo que los cambios sean realmente problemáticos en aplicaciones grandes diseñadas de esta manera [3].

Aunque la arquitectura monolítica presente estos problemas, ha sido y sigue siendo natural construir los sistemas de este modo, ya que toda la lógica para manejar una petición se ejecuta en un único proceso. Esto nos facilita exprimir todo el potencial de un lenguaje de programación, pudiendo dividir la aplicación en clases, funciones y módulos para suavizar el acoplamiento. Como el monolito es nada más y nada menos que una sola unidad, sus pruebas y despliegue son tareas que, manejadas con el cuidado y detalle que merecen, resultan más cómodas de manejar y abordar [3].

A principios de los 2000 surgió una arquitectura que proponía "romper" el monolito, y esta era la Service Oriented Architecture (SOA). Surgió con el propósito de ofrecer reusabilidad de componentes con un bajo acoplamiento entre sí a nivel interno en una empresa. Este enfoque interno que tiene SOA estaba pensado para que diferentes aplicaciones pudiesen hacer uso de un mismo componente dentro de una organización. Esto se conseguía comunicando a estos "servicios" mediante un bus centralizado, con comunicación síncrona y utilizando principalmente XML's. Sin embargo, esta arquitectura se enfrentaba a limitaciones debido a su adherencia a un "monolito externo" como es la organización, además de sobredimensionar y complicar la capa de comunicación que en sistemas como las aplicaciones web pueden lastrar al proyecto y a los desarrolladores, más aún teniendo en cuenta la alta popularidad de estas aplicaciones actualmente [8].

De cierta manera, las dificultades innatas asociadas a la arquitectura monolítica y SOA llevaron al nacimiento de la arquitectura de microservicios. Construyendo las aplicaciones como un conjunto de servicios, independientemente desplegables y escalables y con comunicación asíncrona, conseguimos solucionar los problemas que aquejaban a la arquitectura monolítica y sobrepasar las limitaciones de SOA. Esta independencia nos facilita establecer límites claros entre los diferentes módulos, pudiendo incluso desarrollarlos en diferentes lenguajes de programación. Además, esta separación nos permite trabajar de manera simultánea a diferentes equipos.

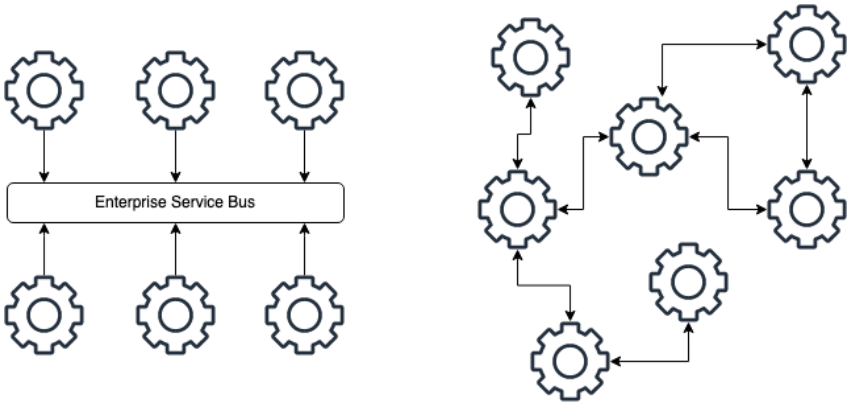

Figura 1. SOA y Microservicios

Profundizando en la arquitectura de microservicios podemos extraer las principales características que la constituyen, más allá de la definición aportada por James Lewis y Martin Fowler. En concreto tenemos las siguientes características [9]. La componetización mediante servicios, la organización alrededor de las capacidades del negocio, la construcción de productos y no proyectos, los endpoints "inteligentes" y los canales "tontos", el gobierno descentralizado, la automatización de la infraestructura y el diseño ante el error.

La componetización mediante servicios en esta arquitectura surge porque definimos los límites de cada uno de estos servicios de manera clara. Actuarán como cajas negras para el resto, siendo ignorantes de los procesos que ocurren dentro de los otros y sólo conociendo lo que sucede dentro de ellos mismos y las peticiones y respuestas que puedan realizar entre ellos [10]. Este principio es en parte el responsable de las grandes tasas de popularidad que ha alcanzado esta arquitectura, debido a la naturalidad de representar los componentes como "piezas físicas" y al estado actual del mundo del software. La adopción de los servicios cloud es cada vez mayor y con esta arquitectura conseguimos una contenerización de los servicios sencilla y clara que podrá aprovechar los recursos eficientemente.

La organización alrededor de las capacidades del negocio está estrechamente relacionada con la anterior característica, ya que la elección sobre qué servicios conformaran los "componentes" de una aplicación no es un proceso arbitrario, sino que está fundamentado en el Principio de Responsabilidad Única [11]. Cada servicio podrá ser desarrollado de manera paralela a los demás, pudiendo desplegarlo y testearlo de manera independiente. Estos desarrollos se pueden llevar a cabo por equipos multidisciplinares, ya que los servicios pueden abarcar desde la UI (User Interface), pasando por el BackEnd hasta las Bases de Datos. Esta forma de construir servicios nos permite solventar el problema que presentó Melvin Conway con su Regla de Conway, que sostiene que los sistemas creados por una organización replicarán la estructura de comunicación de esta [12]. Si estos roles de diferentes disciplinas en vez de estar sectorizados son organizados en equipos multidisciplinares, los servicios serán menos rígidos en su creación y mucho más flexibles y rápidos en llevar a cabo cambio.

La construcción de productos y no proyectos representa un cambio en la filosofía a la hora de desarrollar y mantener una aplicación. Los creadores de este modelo de arquitectura proponen que las aplicaciones desarrolladas no deben ser proyectos temporales, en los que un equipo los construye y una vez completado se traspasa a otro equipo de mantenimiento y el equipo que originalmente la creó es desmantelado [13]. Este enfoque tradicional se ha cambiado por otro más cercano a mejorar la experiencia del usuario, mediante un solo equipo que desarrollará y mantendrá ese software durante todo su ciclo de vida. Esto genera un conocimiento extenso sobre todos los detalles de la aplicación y lo que es más importante, una verdadera relación con el cliente que conseguirá que seamos más efectivos a la hora de resolver todas las situaciones para mejorar las capacidades del negocio.

Los endpoints "inteligentes" en la arquitectura de microservicios nacen de la no/poca confianza que debemos tener en los canales de comunicación. Los microservicios no son una aplicación que ejecuta sus llamadas a métodos en memoria, sino que deben comunicarse en un entorno distribuido. Como tal, las arquitecturas de microservicios deben evitar caer en las falacias de la computación distribuida y no confiar en que las interacciones entre diferentes servicios son fiables, seguras, instantáneas y homogéneas respecto a otras similares o iguales que se hayan realizado previamente [14]. Para ello, serán lo más desacopladas (poco o ningún conocimiento de otros servicios) y cohesivas (cada servicio debe centrarse en ofrecer funcionalidades fuertemente relacionadas) posibles. Junto con el

uso de protocolos ligeros (HTTP principalmente), imitando el funcionamiento de la World Wide Web, se consigue separar la lógica de los canales, recayendo esta en los endpoints y en los propios servicios. Existen ciertas aproximaciones algo más "complejas", como el uso de buses de mensajes ligeros. Un ejemplo es RabbitMQ, un bróker de mensajes que se ubica en los canales y actúa como intermediario sin asumir las falacias de un sistema distribuido, sobretodo desacoplando los servicios a la vez que siguen siendo los que producen y consumen los mensajes [15].

El gobierno descentralizado se ve reflejado en la gran variedad de lenguajes de programación y tecnologías que puede alojar una sola aplicación basada en la arquitectura de microservicios. Esto es así porque las comunicaciones entre los servicios deben de ser independientes de la implementación, haciendo que esta heterogeneidad sea posible. El ideal que se quiere conseguir con esta filosofía es que podamos usar la mejor herramienta o tecnología para resolver cada problema específico, ya que, si sólo disponemos de una solución tecnológica o un lenguaje, como es el caso típico en las arquitecturas monolíticas, probablemente no será el mejor en todas las situaciones que nos encontremos [16]. Gracias a la existencia de estándares en la comunicación entre servicios, como hemos podido ver en el anterior apartado, además de los que establezca cada equipo para su propia aplicación en cuanto a formatos u otros aspectos, esta descentralización ha conseguido grandes resultados como es el caso de Netflix [17]. Es muy importante recalcar que este principio también se aplica en cuanto a la gestión del almacenamiento de datos. En parte esta descentralización viene condicionada por la separación de responsabilidades de los diferentes servicios, por lo tanto, cada uno estará especializado en un tipo de datos concreto. De manera ideal, cada servicio gestionará una base de datos diseñada para tal fin, pero no es raro que varios servicios lleguen a usar la misma base de datos, siempre teniendo en cuenta el Principio de Responsabilidad Única para mantener un buen diseño. Este principio para la administración descentralizada de datos se conoce como Persistencia Políglota [18].

La automatización de la infraestructura es un requisito casi imprescindible no solo en una aplicación basada en microservicios, sino cualquier desarrollo actual. Con la popularización de los servicios en la nube, no ser eficientes en los procesos que van desde el desarrollo hasta la producción de un software puede hacernos gastar demasiados recursos en una tarea que debería ser relativamente sencilla, más teniendo en cuenta la frecuencia tan corta con la que se repiten, debido a los habituales cambios o mejoras en el software. La Integración Continua y el Despliegue Continuo también son factores muy importantes para los equipos hoy en día, ya que avanzar en el desarrollo de diferentes servicios al mismo tiempo es vital para el buen funcionamiento del equipo. Cada servicio dispondrá de su propia "pipeline", que no es más que una serie de pasos para llevar el código desde el desarrollo hasta producción. Los pasos más comunes que siguen estas "pipelines" suelen ser la subida del código a un repositorio, la compilación automática del mismo, una fase de ejecución de tests automáticos, el despliegue en un entorno previo a producción, la realización de pruebas de validación y la promoción a producción [19]. Cuantos más de estos pasos sean completamente automatizados, el equipo ganará mucho en eficiencia y comodidad.

El diseño ante el error responde a la propia naturaleza de sistema distribuido que tiene la arquitectura basada en microservicios. La consecuencia de usar los servicios como los componentes que conforman nuestra aplicación es que debemos prepararnos ante los posibles fallos de estos. Es una desventaja clara ante la arquitectura monolítica, que no presenta toda la complejidad adicional de esta arquitectura. Debido a la naturaleza inesperada de los fallos, se deben establecer mecanismos para que estos no se propaguen y puedan causar una caída total del servicio. El objetivo es mantener una infraestructura que

se “autocure” ante los errores, por ejemplo, reinstanciando un servicio que ha fallado en su ejecución o que un servicio relance peticiones si no han sido escuchadas debido a una caída [20]. Uno de los ejemplos más brillantes de esta filosofía hoy en día es la Simian Army de Netflix, que es una herramienta desarrollada por su equipo para simular caídas del servicio y fallos del sistema de manera aleatoria. De esta manera, consiguen adelantarse a los posibles problemas en caso de que estos sean por causas reales, como por ejemplo la caída de Amazon Web Services en 2015, de la que Netflix solamente se vió afectado solamente por un par de minutos, mientras que otros servicios estuvieron horas caídos [21]. Es una tarea especialmente sensible, donde la automatización no es suficiente y se deben establecer todos los mecanismos posibles para que el equipo mediante acciones manuales y rutinarias pueda conocer y controlar el estado pasado y actual de cada una de las partes de la aplicación.

Conociendo al detalle la base teórica, los fundamentos y problemas que esta arquitectura pretende resolver, y que han impulsado esta arquitectura a sus cotas de popularidad actuales, nos interesa analizar su naturaleza, que es la de un sistema distribuido, y como tal, el mayor reto que nos presenta es la gestión y, sobre todo, el conocimiento del estado del sistema al completo. Por eso, en el siguiente apartado trataremos el estado del arte en cuanto a la observabilidad, investigando sobre las diferentes técnicas que se emplean y analizando de manera crítica cada una de ellas.

3.2 OBSERVABILIDAD EN UN SISTEMA DISTRIBUIDO

En el anterior apartado hemos revisado las características que definen a la arquitectura basada en microservicios, que en su esencia es un sistema distribuido. Esta naturaleza nos aporta una gran cantidad de beneficios, pero viene junto un efecto colateral poco despreciable, que es el aumento de la complejidad en el sistema. Esto afecta a todas las partes que lo involucran, debido a los procesos que ocurren entre estas.

El término observabilidad no es un concepto nuevo, pero recientemente ha ganado mucha importancia en el mundo DevOps. La aproximación DevOps surge en la Agile Conference de Toronto en 2008, donde P. Debois propuso un conjunto de prácticas y procesos para intentar acercar a los equipos de desarrollo y de operaciones, pudiendo dar una respuesta efectiva y conjunta a las demandas del cliente [22]. Los desarrolladores DevOps son los que, con el auge de la arquitectura de microservicios, a mediados de la década de 2010 comenzaron a debatir sobre la observabilidad en el mundo DevOps (y en general en la industria del desarrollo de software), consiguiendo darle la importancia que tiene hoy en día [23].

El origen del término observabilidad lo podemos situar en la ingeniería, concretamente de la teoría de control. El ingeniero Rudolf E. Kálmán estableció en 1960 que un sistema es observable si los estados del sistema y su comportamiento pueden ser determinados solamente conociendo sus entradas y salidas (inputs y outputs) [24].

Esta definición nos deja con un concepto que es eminentemente binario, es decir, si no podemos inferir los estados, el sistema analizado no es observable. Esta naturaleza binaria, en conjunto con la gran complejidad de una arquitectura basada en microservicios hizo que tuviese que revisarse este concepto para dichos sistemas software. Por este motivo, Bryan Cantrill (uno de los creadores de dtrace) en la Obervability Practitioners Summit de 2018 afirmó que esta complejidad hacía que la observabilidad de aplicaciones basadas en microservicios, si asumíamos la definición de la teoría de control, siempre iba a ser cero. Tras esta crítica, presentó una definición que se ha ido estableciendo como la principal referencia para la observabilidad en el mundo del software. Su definición de la obsevabilidad es "la capacidad de permitir a un humano hacer preguntas y obtener respuestas sobre el funcionamiento de un sistema". Cuantas más preguntas y respuestas pueda obtener acerca del sistema, más observable será [25].

Existe cierto debate sobre si el término observabilidad debería haberse mantenido como monitorización, ya que la monitorización, según el IEEE, consiste en supervisar, guardar, analizar o verificar el funcionamiento de un sistema o componente [26]. La monitorización encaja mejor con las técnicas que se aplicaban en la arquitectura monolítica, pudiendo marcar el origen de "los tres pilares de la observabilidad" en técnicas ya establecidas para esta arquitectura, pero no tenía la visión holística que acompaña a la observabilidad.

Actualmente, este concepto se sustenta en llamados "tres pilares de la observabilidad", que son los logs, las métricas y las trazas. A pesar de su aparente similitud, dado que las métricas y las trazas son en realidad una abstracción construida sobre el propio concepto de log, son técnicas con unas diferencias muy marcadas. Los logs son registros inmutables de eventos con una marca de tiempo. Las trazas representan una serie de eventos relacionados temporal y secuencialmente. Las métricas representan mediciones de datos a lo largo de un periodo de tiempo, que requieren de un tratamiento para poder analizarlas e inferir conocimiento de ellas [27]. Estos tres pilares ya eran implementados en sistemas monolíticos para conocer el estado del sistema, saber cuáles eran los resultados de las ejecuciones de los diferentes procesos según su fichero de logs, usando herramientas para coleccionar métricas o analizando la secuencia de una ejecución. El problema y el

consecuente cambio de paradigma surge cuando nos enfrentamos al reto de monitorizar una arquitectura de microservicios, porque no es igual crear trazas de un sistema monolítico que de este tipo. Este problema se vuelve palpable con el simple concepto "en funcionamiento", ya que en una aplicación monolítica podemos saber si está en funcionamiento o no, debido a que es una única entidad. En cambio, en una aplicación de microservicios este concepto es prácticamente imposible de definir debido a la arbitrariedad que existe para poner un límite que diferencie estar "en funcionamiento" de no estarlo (exceptuando la caída total del sistema), debido a la coexistencia de numerosos servicios diferentes. Lo mismo ocurre cuando detectamos que el sistema funciona más lento de lo normal, ya que existen múltiples posibles fuentes que pueden estar generando este problema [28]. Es por estas dificultades a la hora de identificar ciertos problemas o comportamientos por las que para abordar los nuevos retos en esta arquitectura se recogieron técnicas de la monitorización de aplicaciones monolíticas, conformando los tres pilares de la observabilidad [29], con los consecuentes cambios y mejoras a estas técnicas, como veremos más adelante.

La habilidad de poder "preguntar" al sistema es fundamental para el concepto de observabilidad. Esto es de vital importancia para las aplicaciones basadas en microservicios, que pueden variar su despliegue y cambiar sus funcionalidades de manera prácticamente instantánea, debido a su carácter constantemente evolutivo. Ante esta situación, no sólo necesitamos conocer los eventos limitados a errores o anomalías, sino que es fundamental poder "realizar preguntas" al detalle para saber todo tipo de datos (tiempo en ejecución, salud, porcentaje de peticiones de un servicio en concreto…). [30]. Esta capacidad se consigue mediante instrumentación y otras técnicas como el uso de agentes para conseguir datos de alta calidad que nos permiten "saltar" entre los "tres pilares", agregar estos a la hora de lectura (y no en la escritura, para no perder contexto) y hacer uso de análisis estadísticos y juicio de los desarrolladores para poder conseguir las deseadas respuestas sobre el funcionamiento de un entorno emergentemente caótico.

La finalidad última de la observabilidad es saber qué es lo que realmente está sucediendo en nuestra aplicación para verificar su normal funcionamiento, actuar ante aumentos de demanda, extraer conocimientos para aplicar mejoras de la arquitectura o abordar los fallos que puedan ocurrir en los diferentes servicios, como escenarios más típicos [31]. Recientemente también se han propuesto aproximaciones a la observabilidad que engloban los procesos de desarrollo y CI/CD del software, que aportan información sobre estos procesos, su contexto y nos permiten analizarlos para comprenderlos de manera retrospectiva y mejorarlos.

La observabilidad aplicada en arquitecturas basadas en microservicios es un campo muy novedoso y reciente, en el que existen multitud de soluciones, pero tiene retos muy importantes a los que enfrentarse. En un estudio realizado en 2019 [32], en el que se entrevistaron profesionales del mundo del software, se destacaron los siguientes problemas como los más importantes que la observabilidad tenía que abordar: el constante aumento de complejidad de las arquitecturas, la heterogeneidad de los servicios, el necesario cambio en la cultura de las compañías, la avalancha de datos a analizar y la falta de experiencia, tiempo y recursos. Estos problemas, sumados a la cantidad tan variada de herramientas y aproximaciones a la observabilidad, hace que adoptarla no sea nada sencillo. Afortunadamente, fundaciones como la Cloud Native Computing Foundation (CNCF) trabajan para conseguir una estandarización de las diferentes técnicas y apoyan el desarrollo Open Source de las mismas [33].

En los siguientes apartados trataremos las diferentes técnicas para ganar observabilidad sobre un sistema distribuido, analizando las herramientas más importantes y

comparándolas de manera crítica, para finalizar con una reflexión sobre el futuro de la observabilidad.

3.3 TRAZABILIDAD DISTRIBUIDA

La trazabilidad distribuida es una técnica que ha evolucionado de la trazabilidad clásica. La trazabilidad, tanto la clásica como la distribuida, nos da la capacidad de poder conocer todos los pasos dados en un espacio de tiempo para completar una operación [34]. Como hemos descrito anteriormente, esta técnica conforma uno de los tres pilares de la observabilidad y en su esencia es una abstracción construida sobre el propio concepto de log, pero con un enfoque más amplio y continuo sobre el funcionamiento de una aplicación. En aplicaciones monolíticas esta técnica era bastante sencilla de aplicar, ya que las operaciones se ejecutaban en un único sistema de manera interna y controlada.

En cambio, en el mundo de los microservicios la trazabilidad se vuelve muy difusa, ya que son entidades software con un alto grado de complejidad y multitud de interacciones entre sus componentes. La concurrencia temporal de diferentes servicios y versiones de estos hace que la trazabilidad clásica sea una aproximación muy poco útil y confiable para extraer datos y conclusiones sobre los procesos que se dan en el funcionamiento de una aplicación basada en esta arquitectura debido a los complejos caminos que puede recorrer una petición a lo largo de la arquitectura [35].

Perder la capacidad de saber cuál es el comportamiento de nuestra aplicación no es una situación sostenible. La trazabilidad clásica evolucionó para adaptarse, convirtiéndose en trazabilidad distribuida. Surge para suplir esta carencia de la trazabilidad tradicional. Su principal enfoque es mantener un registro de las operaciones de cada servicio, su orden en las interacciones y los tiempos de ambas. De esta manera, podemos obtener los beneficios clásicos de la trazabilidad, pero en un sistema distribuido, pudiendo identificar posibles cuellos de botella, errores en los servicios u otras circunstancias que puedan aquejar el buen rendimiento de una aplicación [36].

Esta técnica tiene su origen junto con la popularización de los sistemas distribuidos y la computación en la nube, aunque sus mayores progresos y mejoras los veremos a partir de 2016, siendo todavía una técnica reciente y novedosa. La primera implementación exitosa a gran escala de la trazabilidad distribuida la encontramos documentada en el paper que publicaron los ingenieros de Google, el conocido como "Dapper", que supuso un antes y un después en esta técnica tras su publicación en 2010 [37]. Este paper, no sólo estableció la mayoría de la semántica que se usa hoy en día, sino que además ofreció una aproximación a la infraestructura necesaria para disponer de una implementación eficiente de esta técnica en un sistema distribuido.

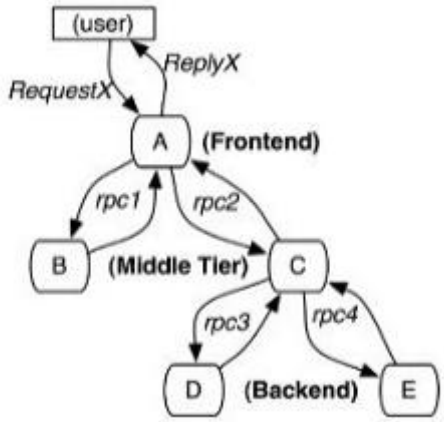

Figura 2. Recorrido de una petición en un sistema distribuido

La figura 2 representa un sistema distribuido simple de cinco nodos. En este sistema, el usuario realiza una petición al sistema, el cuál mediante RPC's (llamadas a procedimientos) entre las partes que lo componen podrá conformar una respuesta a la petición. Para el seguimiento de todo este proceso en el que se intercambia información múltiples veces mediante las RPC's, Dapper usa en conjunto dos partes.

En primer lugar, una parte que se encarga de recoger y unificar toda esta información, de la cuál presentaron dos alternativas, el modelo Black Box y los esquemas de monitoreo basados en anotaciones. El modelo Black Box asumía que la única información necesaria eran los identificadores de mensaje y la timestamp de cada uno, unificando toda la traza de una petición posteriormente mediante técnicas de regresión estadística [38]. En cambio, el modelo de esquemas de monitoreo basados en anotaciones también asume esta información, pero añade aplicaciones o middleware que se encargan exclusivamente de ir guardando las relaciones entre cada una de las peticiones, junto con su timestamp, unificando así toda la traza [39]. Ambos modelos eran propuestas de estudios anteriores, y en base a la práctica en Google escogieron el modelo de esquemas de monitoreo basados en anotaciones debido a la ocasional imprecisión del modelo Black Box (inevitable por el uso de métodos estadísticos) y que las nuevas partes necesarias no suponían un gran impacto en su entorno. Esta decisión, junto con su arquitectura propuesta para implementarla, condicionaron la forma en la que se han desarrollado las herramientas de trazabilidad distribuida, como veremos más adelante.

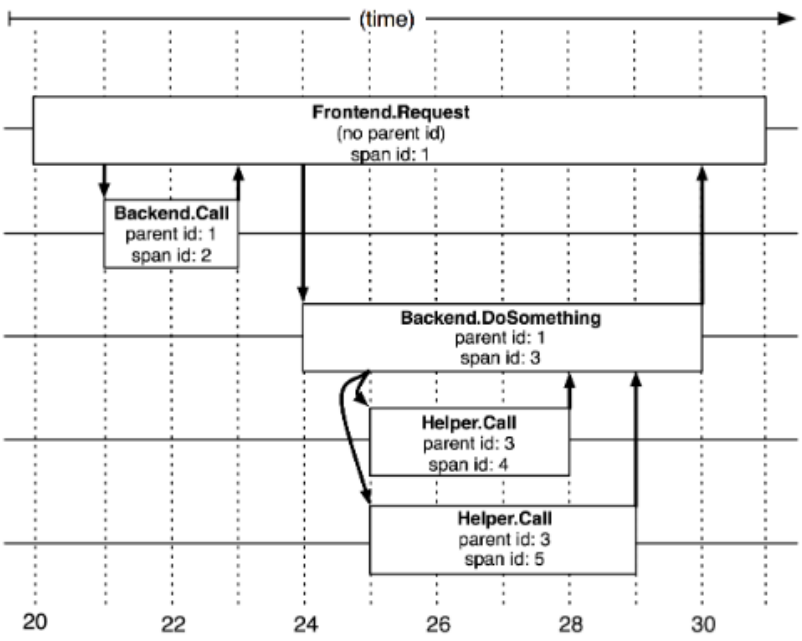

Figura 3. Representación de una traza completa

La otra parte que usa Dapper para conseguir la trazabilidad distribuida es su propuesta de unidades de trabajo. Estas unidades conforman la semántica a la que nos hemos referido anteriormente, que establecieron una base hoy en día vigente para denominar a los conceptos que conforman esta técnica. En este paper, las trazas se representan con la estructura de datos de árbol. Los nodos del árbol se consideran tramos o lapsos (spans), que representan la duración temporal de cada proceso en un servicio. La Figura 3 muestra la representación con esta estructura de datos de los cinco tramos mostrado sen la Figura 2. En el árbol podemos identificar a las trazas hijas ya que disponen del id de su traza "padre" además del suyo propio, pudiendo saber de esta manera todos los pasos dados en un proceso. En definitiva, Dapper establece como unidades con las que trabajará a los tramos, con un nombre legible por humanos, a los numeros de identificación y el posible identificador de padre por cada tramo, además de la representación con la estructura de datos de árbol conformado por "tramos" (que son otro nombre para los nodos clásicos de un árbol).

La arquitectura para la recolección y registro de trazas en Dapper está estructurada en tres fases. En primer lugar, los datos de los tramos se escriben en archivos de registro locales de cada host que genere trazas. En segundo lugar, se extraen dichos ficheros con una infraestructura de recopilación que unifica las trazas hijas y padres y, finalmente se escriben en una celda en los repositorios de Dapper Bigtable (sistema de almacenamiento de datos de alto rendimiento creado por Google). Una traza se presenta como una única fila de Bigtable con cada columna correspondiente a un tramo. Esta arquitectura presentaba una latencia media para generar una traza completa de 15 segundos, una velocidad récord para la época.

Todas las propuestas de Dapper tuvieron un gran impacto en los avances de esta técnica, dando origen a la base de las futuras herramientas. La primera herramienta que se basa en las propuestas de Dapper es Zipkin, cuyo lanzamiento se remonta a 2012, cuando Twitter lo distribuyó como Open Source. Su principal objetivo era aplicar esta técnica para identificar los problemas de latencia en arquitecturas distribuidas, especialmente en las arquitecturas basadas en microservicios, coleccionando datos y permitiendo su visualización [40].

La propuesta de arquitectura de Zipkin es relativamente similar a la de Dapper, ya que los hosts (o servicios) se encargan de generar y enviar las trazas al colector medante protocolo HTTP, donde serán analizadas y, posteriormente, almacenadas en una base de datos. Además de estas partes (o fases, como se llamaban en Dapper), disponemos de una API y una UI para poder visualizar las trazas y extraer conclusiones [41].

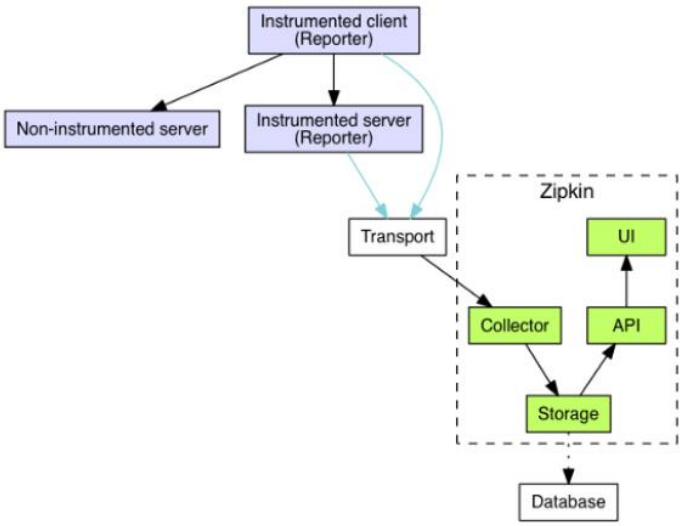

Figura 4. Arquitectura de Zipkin

Zipkin requiere que cada servicio genere sus propias trazas. Para ello, dispone de multitud de librerías con soporte oficial y de la comunidad Open Source para instrumentar los servicios de manera que puedan generar las trazas y enviarlas al colector [42]. Estas librerías nos permiten implementar en nuestro código de manera sencilla toda la instrumentación necesaria. Cada traza será generada en base a un modelo de datos propio que adapta la semántica propuesta por Dapper, conteniendo información sobre su marca de tiempo, su ID de traza, su ID de traza padre y diversos metadatos [43]. Dicho modelo de datos también se usará para almacenar las trazas, tanto en Bases de Datos NoSQL como Casandra o ElasticSearch o en bases de datos relacionales como MySQL. Al ser creadas por el propio host, pueden tener un cierto impacto en su funcionamiento, haciendo que ese servicio pierda algo de eficiencia. Además, si una arquitectura es muy grande, la existencia de un único colector puede actuar como un cuello de botella importante. Es por esto por lo que Zipkin recomienda muestrear las trazas para estos casos, ya que almacenar todas las trazas podría impactar en el rendimiento [44]. Otra solución a esta problemática es crear un balanceador de carga propio para gestionar el tráfico a varias instancias del colector de Zipkin, lo cual requiere de un esfuerzo extra [45].

Debido a la necesidad de que cada servicio se encargue de enviar sus propias trazas, Zipkin es una herramienta de trazabilidad distribuida que se clasifica según el tipo "Transferencia de Traza Directa" [46]. Durante su desarrollo y maduración, Zipkin ha conseguido mantenerse relevante, con un aporte continuo de la comunidad como podemos observar en su repositorio Open Source [47]. En este proceso Zipkin ha adoptado cambios significativos debido a los esfuerzos por la estandarización de esta técnica. Entre ellos, el más notable fue marcado por la aparición de OpenTracing.

La gran variedad de servicios que se pueden instrumentar junto con su diferencia en los lenguajes, además de las diferentes herramientas que fueron surgiendo después de Zipkin para disponer de trazabilidad distribuida supuso un problema creciente que afectaba directamente al funcionamiento de esta técnica [48]. Ante esta situación, surge

OpenTracing en 2016, con el objetivo principal de permitir una instrumentación sencilla de las aplicaciones y sin que la decisión por implementarla con una herramienta nos ate permanentemente a ella.

OpenTracing propone una especificación para API's, además de alojar frameworks y librerías que implementan dicha especificación. Tan importante fue este paso que rápidamente fue adoptado como proyecto en la CNCF (Cloud Native Computing Foundation) en 2016, que es un proyecto de la Fundación Linux que aboga por un alineamiento de la industria con los progresos y evoluciones de las tecnologías más destacadas de la computación en la nube. Ponen un principal hincapié en la neutralidad en cuanto a los proveedores de las herramientas, al igual que OpenTracing [49]. Es importante destacar que la CNFC no es un organismo oficial de estándares, pero la comunidad internacional de desarrollo de software, sobretodo la OpenSource, apoya esta iniciativa para conseguir una mayor homogeneidad y aproximarse a una estandarización de las tecnologías.

La especificación que ofrece OpenTracing está construida alrededor de dos conceptos básicos, los tracers (trazadores) y las spans (tramos o lapsos), Los tracers son los que crean las trazas, pudiendo inyectar o extraer el contexto de los spans, generando así la secuencia de la traza. Los spans son las unidades con las que trabajan los tracers, pudiendo operar sobre ellas para añadir etiquetas, ponerlas nombre o añadir referencias a otras spans [50]. Además de presentar esta especificación para la creación y manipulación de trazas, OpenTracing presenta un modelo de datos basado en la estructura de datos de Grafos Dirigidos Acíclicos [51], que evoluciona respecto a árbol que se usaba en Dapper y en Zipkin. Aunque Zipkin era un proyecto maduro y el más popular en esa época, decidió apostar por OpenTracing y a finales de 2017 empezó a soportar la especificación OpenTracing en algunas de sus librerías.

Como primera aproximación a una homogeneización de esta técnica, OpenTracing tuvo bastante éxito, soportando implementaciones en los lenguajes más populares (Java, Python, JS, Go, etc.), además de preparar el camino para nuevas herramientas y futuros proyectos.

Entre esos proyectos surge Jaeger en 2017, cuando Uber decide convertir el proyecto en Open Source. Este proyecto nace como conjunción de todos los esfuerzos anteriores, inspirándose en Dapper y en Zipkin, además de implementar la especificación OpenTracing desde su comienzo [52]. Teniendo como caso de éxito a la propia Uber, Jaeger ganó rápidamente popularidad aún siendo un proyecto muy poco maduro.

Los ingenieros de Uber comenzaron a desarrollar Jaeger de forma totalmente modularizada, sustituyendo gradualmente los componentes de Zipkin que usaban en sus sistemas hasta que terminaron creando Jaeger como una sola herramienta. Las principales motivaciones detrás de este reemplazo fueron que Zipkin no soportaba la especificación OpenTracing y que el modelo de datos basado en árboles resultaba mucho menos versátil que los grafos acíclicos dirigidos [53]. Este proceso de creación modular tiene dos efectos colateral particularmente beneficiosos, que es la gran capacidad de escalado de la que dispone Jaeger, al no presentar puntos únicos de fallo, y la retrocompatibilidad con Zipkin, permitiendo a los usuarios poder cambiar de Zipkin a Jaeger con relativamente poco esfuerzo [54]

La arquitectura de Jaeger supone un importante cambio en la filosofía respecto a Zipkin y a Dapper, decidiendo apostar por la abstracción en el proceso de envío de las trazas de un host. Este nuevo diseño se conoce como "Transferencia Basada en Agente Instalado" [46] y nos permite abstraer todo el enrutamiento de trazas y descubrimiento de hosts.

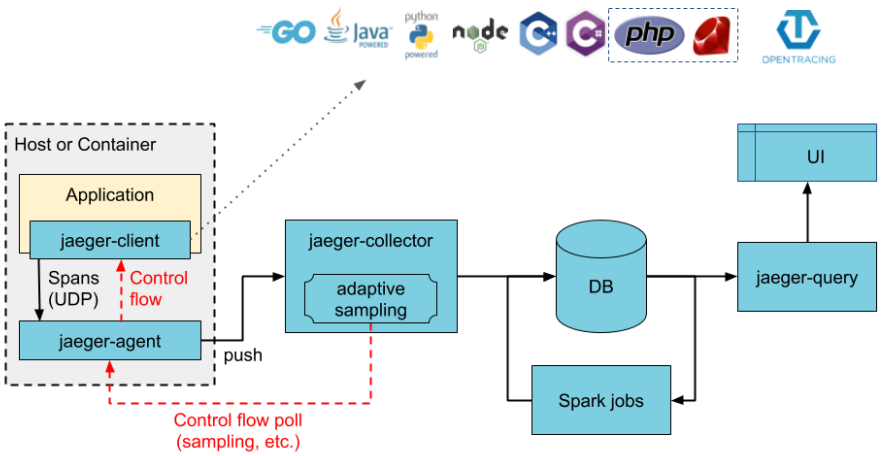

Figura 5. Arquitectura de Jaeger

En la Figura 5 se representa la arquitectura de Jaeger, compuesta por el cliente, el agente, el colector, la base de datos, un sistema para realizar queries y la UI. El elemento más novedoso de su arquitectura es el agente, que es un demonio que se encarga de escuchar las trazas que recibe del cliente mediante UDP y que serán enviadas en lotes al colector. Este agente debe ir junto con la aplicación o servicio que se desee monitorizar, ya que es el encargado de materializar la abstracción que hemos mencionado anteriormente, encargándose él mismo del envío al colector de las trazas. Entonces tendremos en el mismo host el agente y el cliente, donde este último sí que depende de la implementación de la aplicación. [55] Los clientes, aunque se mantengan en la arquitectura oficial de Jaeger e incluso implementen la especificación OpenTracing, han sido deprecados recientemente y se incita a utilizar la especificación OpenTelemetry, disponiendo de librerías para instrumentar servicios en una gran multitud de lenguajes, pero esto lo trataremos más adelante

En cuanto a las otras partes de la arquitectura, Jaeger es relativamente similar a Zipkin, pero con ciertos matices que hacen a Jaeger más seguro en sistemas relativamente grandes. Jaeger surge como una herramienta para trazar un sistema muy grande como lo es Uber, así que sus ingenieros lo dotaron de una arquitectura enfocada a disponer de una buena escalabilidad. Tanto es así que en la propia documentación se muestra otro esquema de arquitectura, dispuesto de componentes adicionales. [55]

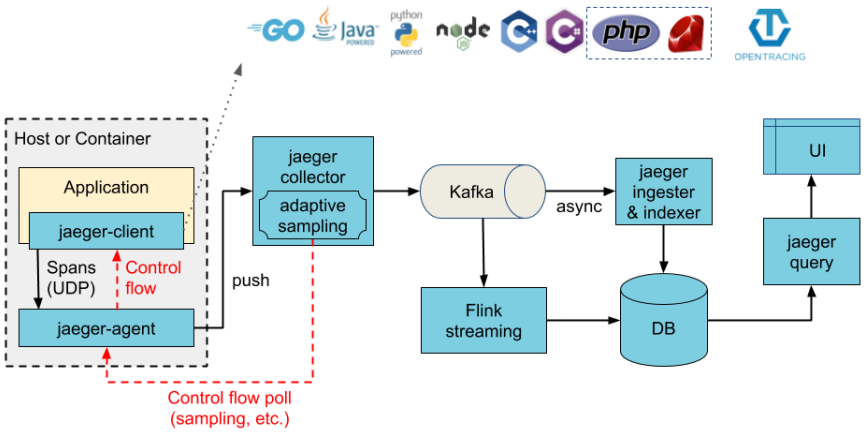

Figura 6. Arquitectura de Jaeger escalable con Kafka

Aparece un sistema de transmisión de mensajes/datos como lo es Kafka, enfocado a recibir mensajes desde múltiples instancias de colectores que puedan estar desplegadas, y enviarlos al "ingester & indexer", otra pieza nueva, que se encargará de consumir el tópico de Kafka de manera asíncrona y escribir en base de datos. Estas nuevas piezas dotan al sistema de una capacidad mucho mayor para recoger datos de sistemas de un tamaño considerable que si no dispusiésemos de ellas, siendo soportadas y disponiendo de documentación para su uso de forma oficial [56].

Debido los cambios y propuestas que Jaeger puso sobre la mesa, ganó popularidad rápidamente y en el mismo año que fue liberado como OpenSource (2017) empezó a ser soportado como proyecto oficial en la CNCF [57]. La novedad de Jaeger y su arquitectura modernizada pero más compleja que la de Zipkin hizo que cierto sector de la comunidad no apostase por esta herramienta, prefiriendo soluciones tradicionales y maduras como lo era Zipkin. Con el tiempo, Jaeger ganó madurez y sus propuestas se han establecido como un un caso de éxito, haciendo que Zipkin siguiese sus pasos para mejorar su modelo de datos [58].

Con la popularidad de los microservicios y las herramientas de trazabilidad distribuida creciendo año tras año, en 2019 se anunció OpenTelemetry. Este proyecto nace de la fusión de los dos proyectos más populares que apostaban por la estandarización en cuanto a esta técnica, OpenTracing y OpenCensus. Este último era un proyecto Open Source de Google, definido como una colección de librerías específicas para cada lenguaje para instrumentar aplicaciones, recoger estadísticas y exportar datos a un backend soportado (Zipkin, Jaeger, AWS X-Ray, Elasticsearch, etc.) [59]. Tanto OpenCensus como OpenTracing buscaban hacer más accesible la trazabilidad distribuida para los usuarios, pero disponer de dos "estándares" que competían entre sí resultó ser más perjudicial que beneficioso en la comunidad. Aunque fuesen competidores, las aproximaciones de ambos no eran contradictorias, y esto fue lo que llevo a ambos equipos a fusionarse en un solo proyecto, OpenTelemetry [60].

Durante la mayor parte de 2019, los equipos de OpenTracing y OpenCensus trabajaron de manera conjunta para crear OpenTelemetry, con el objetivo de unificar ambos proyectos y disponer de una única entidad que disponga de especificación de datos, de instrumentación y de infraestructura no sólo para la trazabilidad distribuida, sino también para logs y para métricas [61]. Este proyecto consiguió rápidamente destacar entre la comunidad debido al éxito previo de los participantes y empezó a ser soportado como proyecto oficial de la CNCF en 2021 [62].

OpenTelemetry ofrece API's, un SDK, instrumentación e integraciones para multitud de lenguajes, siguiendo la arquitectura propuesta por Jaeger de "Transferencia Basada en Agente Instalado". Además, disponemos de una documentación muy extensa para implementar los casos de uso más típicos, con despliegues en Docker y en Kubernetes. Todo el esfuerzo de OpenTelemetry está orientado a que las trazas sean generadas y recogidas en base a unas especificaciones comunes, dejando el trabajo de almacenamiento, manipulación y análisis a los vendedores como pueden ser Jaeger o Zipkin, consiguiendo así que la decisión por una u otra herramienta no tenga absolutamente ningún impacto en cuanto a estos dos procesos (generación y colección de trazas) [63].

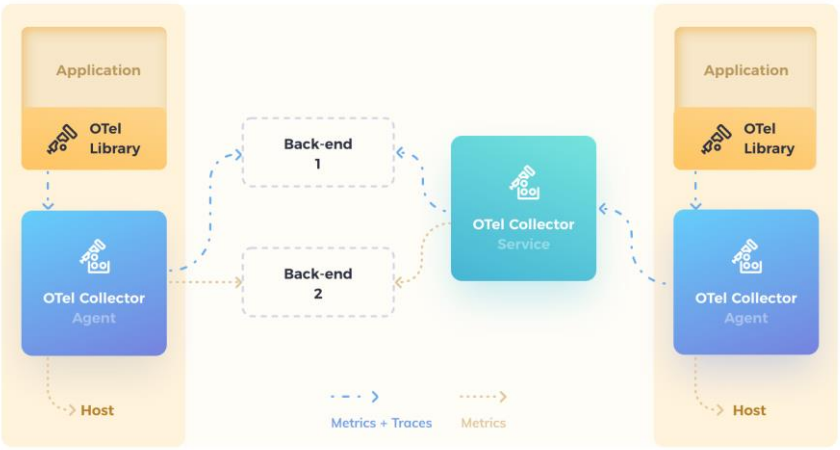

Figura 7. Arquitectura de OpenTelemetry

Respecto a las novedades más actuales, las herramientas están adaptándose a los estándares de OpenTelemetry, además de poco a poco ir convirtiéndose en herramientas de un propósito más general y abarcando técnicas diferentes a la trazabilidad distribuida, como pueden ser las métricas y los logs. Por ejemplo, herramientas como Hypertrace soportan actualmente OpenTelemetry, y Zipkin y Jaeger también están trabajando en este camino. La utilidad de esta técnica para ganar observabilidad en un sistema es totalmente indiscutible, y el avance en los últimos 10 años ha sido muy notorio, con una comunidad cada vez más activa, como se puede ver en las estadísticas de la CNCF, dado que OpenTelemetry es el segundo proyecto más activo de los que apoyan, sólo por detrás de Kubernetes [64].

La trazabilidad distribuida es, de los tres pilares de la observabilidad, el que más avances ha visto en los últimos años. Es especialmente útil para ganar conocimiento sobre los procesos complejos que se dan en el sistema que requieren de varios pasos por diferentes servicios, pudiendo identificar caminos críticos, puntos donde el sistema falla más de lo habitual o las diferencias de tráfico y rendimiento entre los servicios en el sistema. Pero esta técnica no es suficiente para disponer de una buena observabilidad por si sola, ya que identificar las causas de los errores puede ser imposible si no disponemos de otras técnicas, por ejemplo, logs que nos proporcionen contexto de ejecución o métricas históricas sobre el funcionamiento de un componente. Por eso recalcamos la importancia de la trazabilidad distribuida, pero no debemos perder el foco de que tiene que ser complementada con los otros dos pilares para ganar una buena observabilidad.

3.4 LOGS

Una de las técnicas más sencillas y antiguas para obtener conocimiento sobre el funcionamiento de un sistema son los logs. El tipo de log más conocido es el log de eventos, usado ampliamente en el desarrollo de software, incluso siendo la primera interacción de algunos desarrolladores con la industria (¡Hola, mundo!), que será al que daremos más importancia en este estado del arte sobre los logs para ganar observabilidad. También existen otros tipos de logs, como logs de servidores, change logs, o system logs [65], pero todos tienen un mismo concepto que los estructura como log.

Un log es una secuencia de registros ordenados por tiempo de manera secuencial, a la que se le van añadiendo progresivamente más de estos durante el funcionamiento del sistema y la sucesión de ciertos eventos que son los que los generan [66]. Habitualmente el log es referido como un solo registro o evento y el fichero de logs será la secuencia ordenada de estos. Un log debe disponer de datos que lo contextualicen y una marca de tiempo para conocer cuándo fue generado, de manera que el fichero de logs estará desacoplado de un reloj físico propio. La cantidad de metadatos para contextualizarlo puede ser muy diversa, desde qué componente ha generado el log hasta información programada por el propio usuario para un evento en concreto que registrará ese log.

La generación automática de logs es una técnica prácticamente estándar en la industria desde hace varias décadas, por ejemplo, cuando Java en 2002 implementó su API para logs [67], pero a la hora de aprovechar la utilidad a esta técnica también hay que exprimir el potencial que nos permite esta técnica creando logs propios en el desarrollo. Gracias a estos dos tipos de logs, podemos extraer información como el rendimiento de procesos, identificación de bugs o información sobre el estado actual del sistema. Es por esto que, además del estado del arte que vamos a revisar sobre esta técnica, seguir buenas prácticas para que el equipo de desarrollo construya buenos logs durante el desarrollo (como establecer diferentes niveles de logs) [68], ya que si no disponemos de logs informativos y comprensibles esta técnica pierde bastante capacidad para ayudarnos a ganar observabilidad sobre un sistema.

En sistemas monolíticos esta técnica es muy sencilla de utilizar, ya que disponemos únicamente de dos niveles de complejidad posibles, la no concurrencia y la concurrencia de un sistema [69]. En cualquiera de estos niveles, cuando ejecutamos una aplicación monolítica tipo, por ejemplo, desarrollada en Java, disponemos de un único fichero en el que se volcarán todos los logs generados con su marca de tiempo y sus datos. En lenguajes interpretados como Python estos logs son generados en la consola, pero es muy sencillo conseguir que se vuelquen en un fichero de logs al uso. El concepto clave tras las aplicaciones monolíticas respecto a esta técnica es la existencia única del fichero de logs [70]. El análisis clásico de este fichero para identificar problemas o determinados eventos suele ser abordado por un humano, analizándolo y extrayendo conclusiones.

Esta técnica sin embargo se vuelve muy difusa y difícil si la aplicamos en sistemas distribuidos, que serían el tercer nivel de complejidad posible de aplicación de esta técnica [69]. La aparición de la arquitectura basada en microservicios y los contenedores han presentado grandes avances y aportan múltiples beneficios para el desarrollo y despliegue del software, pero también conllevan ciertas desventajas. La desaparición de la unidad que es el monolito implica que los datos de una aplicación estarán repartidos por múltiples y diferentes servicios, al igual que la generación de logs, que se verá diseminada en múltiples ficheros de logs independientes. Además de esta separación, los contenedores nos presentan otras barreras que esta técnica debe superar, como lo es que pueden ser creados y destruidos, llevándose con ellos todos los datos que puedan contener en su instancia o

que existan múltiples instancias o versiones de un mismo servicio, siendo necesario identificar quién ha generado cada log de manera inequívoca.

Esta serie de desafíos que presenta esta técnica en sistemas distribuidos y, principalmente, en las arquitecturas basadas en microservicios, han causado que esta técnica evolucione para superarlos y adaptarse a la nueva manera de construir aplicaciones software. El presente estado del arte recoge las prácticas, arquitecturas y avances más importantes que se han dado en este aspecto. Para disponer de una buena implementación de esta técnica en primer lugar es completamente necesario hacer hincapié en una serie de buenas prácticas [71] que los desarrolladores han ido creando y poniendo en común según esta técnica se ha ido desarrollando, no siendo un estándar oficial, pero si unas guías para conseguir una implementación útil.

La fundamental de estas buenas prácticas que vamos a abordar es la estructuración de logs. Cuando la información no tiene un modelo de datos predefinido o no está organizada en torno a una manera se considera que son datos desestructurados. Desafortunadamente, logs son por defecto datos de texto desestructurados y corre por nuestra cuenta dotarlos de una estructura propia. Las ventajas de aplicar un modelo para estructurarlos, pudiendo ser tan sencillo como marca de tiempo, tipo de log, mensaje informativo y usuario puede marcar la diferencia y mejorar tareas tan comunes como buscarlos, analizarlos o parsearlos (separar datos en diferentes partes para su manipulación y almacenamiento), disminuyendo así la posibilidad de causar sobrecarga (overhead) en el almacenamiento [72].

Otra buena práctica es hacer que los logs sean informativos. Esta buena práctica está bastante relacionada con la anterior, pero se enfoca sólo en el mensaje que acompaña al resto de la estructura de un buen log. En este mensaje se debería informar como mínimo del propósito de la operación, y de manera ideal la posible fuente del error [73]. Esta buena práctica es muy beneficiosa en situaciones de errores graves, para ayudar a identificar el foco del problema e idealmente solucionarlo rápidamente.

Las anteriores buenas prácticas también son aplicables en sistemas monolíticos, pero son imprescindibles a la hora de usar esta técnica en un sistema distribuido. Es por esto por lo que los logs se sustentarán en estas prácticas además de en la arquitectura. El fundamento detrás de la arquitectura es sobrepasar la descentralización y diseñar un sistema centralizado de logs, que nos permitirá una búsqueda y un análisis mucho más sencillo que tratar uno por uno los logs distribuidos [74]. Un diseño básico para aplicar esta técnica en un sistema distribuido se dividirá en tres partes, los nodos "colectores", los nodos "agregadores" y el almacenamiento [75]. En primer lugar, los nodos colectores se componen de un servicio al uso que realiza sus funciones y genera logs, que serán recogidos por un agente que acompaña a este servicio en su contenedor (como en Jaeger) y los enviará a un nodo "agregador" (pudiendo ser tan sencillo como un solo cluster Kafka). Este nodo actuará para desacoplar el almacenamiento de los nodos, consiguiendo recibir datos de múltiples fuentes y enviándolos al almacenamiento, donde serán guardados para su posterior uso.

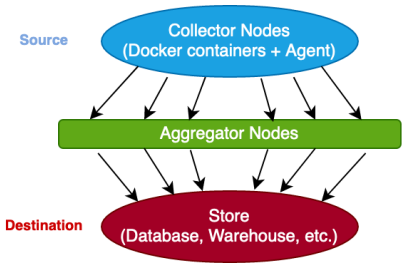

Figura 8. Arquitectura básica para recolección de logs en un Sistema Distribuido

Además de estas piezas básicas en la arquitectura, se proponen ciertos patrones en función de las necesidades del sistema en el que se implanta [76]. La arquitectura que hemos descrito al enfrentarse a un escalado de los nodos colectores sufrirá problemas de rendimiento en el agregador ya que estará conectado a demasiados nodos a la vez. Estos patrones surgen para mitigar este problema. El primero de estos patrones es la agregación por origen, que se consigue instanciando varios agregadores secundarios, que consumirán logs de un conjunto determinado de servicios para posteriormente reenviarlos a un agregador primario, que será el encargado de almacenar estos logs, mejorando el rendimiento del agregador primario al repartir la carga de trabajo y reducir el número de conexiones. Otro de los patrones para enfrentarse al escalado es el balanceo de carga, muy conocido en el desarrollo de software, pero de novedosa aplicación en esta técnica. Con un balanceador de carga podemos repartir el tráfico proveniente de los colectores a las diferentes instancias de agregadores y mitigar los impactos de un tráfico muy alto en el rendimiento de las partes de la arquitectura. Estos patrones deben ser valorados, junto con la opción de muestreo de logs menos relevantes (por ejemplo, las HTTP 200), en función de los recursos y la capacidad de nuestro sistema, ya que si una implementación básica es viable debido al tamaño del sistema y los recursos (CPU y memoria principalmente) disponibles, no interesaría demasiado complicar la arquitectura con elementos innecesarios.

Se han explicado las buenas prácticas en este orden para poder extraer la conclusión de que para esta técnica es indispensable comenzar desde abajo, con unos buenos logs estructurados e informativos, que serán recogidos por una arquitectura eficiente para poder contar con la última buena práctica, que será hacer los logs "buscables" [77]. Esto significa que podamos aprovechar la estructura del modelo de datos de los logs para poder realizar búsquedas y análisis, facilitando principalmente tareas tan críticas como el trabajo de identificación de errores y los análisis de rendimiento.

Existe cierto debate sobre si en la arquitectura deben encargarse del parseo los agentes o nodos dedicados a ello después de la colección de estos, ya que la carga adicional que se impone a los contenedores de los servicios puede ser demasiado grande en determinados escenarios, y es más adecuado asegurar un funcionamiento estable del sistema y dedicar otros recursos de manera exclusiva al parseo de estos datos [78]. Esta implementación puede ser tanto offline como online, es decir, siendo las implementaciones más comunes un procesamiento de lotes de logs (offline) o una transmisión directa de cada log parseado (online) [79].

En estos últimos años se ha dotado a algunas herramientas que usan esta técnica de características más allá de la recolección de logs. Entre otras, destaca el uso del "Machine Learning" para aplicar modelos de detección de anomalías para identificar logs que podrían ser importantes o identificar situaciones en las que un sistema podría acabar en error. A pesar de estos grandes avances, son todavía aplicaciones muy novedosas en sistemas productivos, pero la literatura indica un futuro muy prometedor en esta línea [80].

Las soluciones para disponer de esta técnica en un sistema distribuido suelen ser aplicadas gracias al uso de herramientas que se basan en todas las buenas prácticas que hemos descrito y permiten una implementación más sencilla que si un equipo de desarrollo tuviese que construir todas las partes de esa infraestructura.

Una de las herramientas más populares es Graylog (Open Source), que dispone de agentes para acompañar a los servicios y un servidor que nos permite visualizar los logs, con diferentes paneles y alertas personalizables, además de permitirnos realizar búsquedas en función a diferentes parámetros. Esta herramienta utiliza un modelo de datos propio llamado GELF para estructurar los logs, muy conveniente ya que es muy sencillo de implementar y nos permite extraer todas estas ventajas de esta técnica [81].

Logstash también es una herramienta que aplica esta técnica, al principio limitada por su compatibilidad con el stack ELK (ElasticSearch, Kibana), pero superada posteriormente con los esfuerzos de la comunidad OpenSource. Logstash y el stack ELK son muy reconocidos por su excelente capacidad de escalado, además de su soporte nativo de OpenTelemetry, pero sus virtudes se han visto recientemente empañadas por su relicenciación a no-OSS. [82]. Al contrario que Graylog, Logstash es una herramienta con un propósito de análisis mucho más general que acepta una gran variedad de formatos para los logs, siendo el más común JSON, además de aprovechar las capacidades de OpenTelemetry en cuanto a trazabilidad y métricas. [83]

La última herramienta que analizaremos en este estado del arte es FluentD. En primer lugar, es importante destacar que es la única herramienta que la CNCF ha adoptado como proyecto oficial en cuanto a la gestión de logs [84]. Uno de los principales objetivos de este proyecto es unificar la colección y el consumo de datos (en concreto logs) para permitir un mejor uso de estos. Esto se traduce en una herramienta que permite una integración con prácticamente cualquier sistema que produzca logs y servirlos a la herramienta de análisis deseada. Este último punto es importante recalcarlo, ya que FluentD no se encarga de analizar ni almacenar los datos, pero posee extensa documentación para las integraciones con las herramientas más populares como MongoDB o Elasticsearch. Casos de éxito de la implantación de esta herramienta podemos verlos en Google o en Amazon Web Services, lo que aumentó en gran medida su popularidad [85].

Esta técnica es tremendamente útil para identificar la raíz de un problema, y en un sistema distribuido esto es fundamental para un buen funcionamiento de los servicios que se ejecutan en el ecosistema de una aplicación, pero esta técnica por sí sola no nos otorga una observabilidad completa. Encontrar un error es una tarea que consume bastante tiempo, y si no disponemos de otras técnicas para ganar observabilidad, identificar un problema que se ha propagado por diferentes servicios puede ser una tarea tremendamente complicada. Es por eso por lo que, los logs dentro de los tres pilares de la observabilidad es la técnica más "fina" de todas, ya que requiere de un gran esfuerzo en una buena implementación de estos para aprovechar todo su potencial, pero es la que más tiempo consume y necesitamos de tanto la trazabilidad como las métricas para suplir los puntos débiles de esta técnica.

3.5 MÉTRICAS

Las métricas son una técnica con un largo recorrido en el mundo del software, debido a su enorme utilidad tanto en sistemas monolíticos como distribuidos. Estas son una representación numérica de datos, principalmente con el propósito de determinar el adecuado o erróneo comportamiento de un software durante un periodo de tiempo. Esto supone que podamos obtener información muy valiosa sobre el histórico de rendimiento del sistema, contar o medir un valor determinado y establecer alertas [86]. Una de las aplicaciones más relevantes para los sistemas distribuidos y en concreto para la arquitectura de microservicios es el uso de análisis estadísticos sobre estas métricas, permitiéndonos determinar el comportamiento y el rendimiento del sistema, consiguiendo ganar observabilidad en estas arquitecturas tan intrínsecamente complejas [87].

Los estados en los que un sistema puede encontrarse son muy variados, casi infinitos si trabajamos con un sistema distribuido, como hemos mencionado anteriormente. Las métricas dependen en gran medida de la capacidad de análisis humano, [88] permitiéndonos extraer conclusiones de estos datos que van mucho más allá de "en funcionamiento" o "no funcionando". Esos estados suelen ser recogidos con los "healthchecks", que, aunque forman parte de esta técnica y son extensamente usados, son una visión muy simplista y reducida de las posibilidades que nos ofrecen las métricas. Si obtenemos datos sobre el uso de los recursos, rendimiento de los servicios, picos inusuales u ociosidad podremos establecer estrategias para escalar/desescalar tanto vertical como horizontalmente, aprovechando al máximo las posibilidades que nos ofrece esta técnica [89].

Un aspecto fundamental de las métricas es su relación necesaria con el instante de tiempo de los datos que registran. Por esta razón se popularizó rápidamente el uso de Bases de Datos de Series de Tiempo (TSDB). Las TSDB surgieron para registrar datos financieros y el intercambio de stocks en bolsa, pero su uso llegó rápidamente a esta técnica debido a la sinergia natural entre ellas. Las TSDB nos permiten registrar mediciones o eventos, pudiendo comprimirlos posteriormente, reduciendo así la muestra para conseguir un almacenamiento eficiente y estable para poder cuantificar el cambio a lo largo del tiempo [90]. Al contrario que los logs, la sobrecarga ("overhead") en el almacenamiento es constante y no es directamente proporcional al tráfico, pero sí que depende del número de variables que tenga cada métrica generada por el sistema, generalmente compuestas por dos partes principales, un nombre propio y un conjunto de variables clave-valor [91]. La colección de conjuntos de variables posibles para una métrica se define como "cardinalidad" y de ella dependerá en gran medida la calidad y eficacia de esta técnica, ya que una alta cardinalidad nos permitirá medir y evaluar más estados e implicaciones de cada métrica.

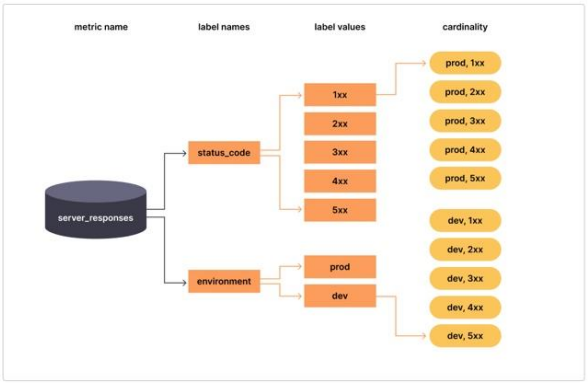

Figura 9. Cardinalidad de una métrica

Sin embargo, esta es un arma de doble filo, ya que a mayor cardinalidad las TSDB empeoran notablemente su rendimiento, degradando el rendimiento y en casos muy graves pudiendo causar errores en el almacenamiento o fallos en el sistema [92]. Más adelante, trataremos con más detalle este aspecto de las métricas y cuáles son las estrategias para mitigar los efectos negativos de una alta cardinalidad.

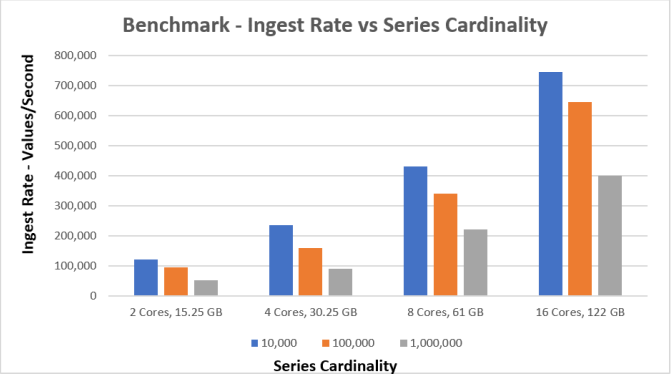

Figura 10. Benchmarks sobre rendimiento en diferentes hardware de series temporales de diferente cardinalidad

Si queremos trazar un origen de esta técnica podemos remontarnos hasta los años 60 con los inicios de la industria del software, donde las limitaciones de recursos hardware suponían un reto enorme para los desarrolladores. En cierto modo esta problemática sigue existiendo hoy en día, siendo fundamental conocer el uso de los recursos para mejorar los procesos, aunque rápidamente monitorizar únicamente el hardware demostró no ser suficiente y se introdujeron gradualmente mejoras que permitían monitorizar aspectos como la experiencia del usuario o el rendimiento del código [93]. Las métricas han estado muy unidas al concepto de monitorización e incluso su desarrollo hasta hace pocos años ha ido ligado al avance de ese concepto, encarnado en las Aplicaciones de Monitorización de Rendimiento (APM). Es por esta razón que en este estado del arte abordaremos la evolución de esta técnica junto con las APM, haciendo hincapié únicamente en las características puramente relacionadas con las métricas y su visualización, ya que estas han evolucionado y crecido hasta llegar a incorporar los logs y la trazabilidad distribuida entre sus capacidades, pero nos conviene conocer sus avances y la maduración que dieron a las métricas como técnica para ganar observabilidad.

Una APM, de manera similar al concepto de la observabilidad, busca racionalizar el funcionamiento de un sistema para que un humano pueda comprenderlo a través de, principalmente, "dashboards" y métricas que, esencialmente, recogen el uso de CPU, la ratio de errores, el tiempo de respuesta o la disponibilidad de la aplicación, entre otras [94]. A pesar de sus similitudes, es vital establecer claramente las diferencias entre ambos conceptos. Anteriormente hemos comentado que la monitorización se asocia más con el monolito, y esto no es casual ya que tanto las APM como las métricas surgieron en este tipo de arquitecturas, recopilando y analizando los datos de un único sistema. Esto se trasladó a los sistemas distribuidos, con ciertas limitaciones y desafíos que hoy siguen enfrentando, como la alta cardinalidad. En este punto es donde la observabilidad entra en juego y se diferencia de la monitorización, permitiéndonos agregar datos de diferentes fuentes y mediante la optimización de esta técnica y con el uso de algoritmos basados en la teoría de control, podemos acercarnos a la visión holística del sistema.

Siendo conscientes de esta diferencia, podemos entender que exista cierto debate sobre si la monitorización en realidad es un componente más de la observabilidad, y viendo la gran relación que existe entre las métricas y las APM, podríamos establecer que la monitorización es un prerrequisito para la observabilidad, puesto que nos suministrará una gran cantidad de información fundamental para que podamos alcanzar la observabilidad del sistema [95].

La etapa primeriza de las APM, conocida como APM 1.0, comenzó en la década de los 90 junto con las aplicaciones de 3 niveles. Estas aplicaciones son sistemas monolíticos por definición, contando con una capa de presentación, una capa de aplicación (o negocio) y una base de datos. Esta generación de APM proporcionó a los desarrolladores las métricas y estadísticas que generaban sus sistemas, pero con dificultades en la implementación y mantenimiento debido a la rigidez de instrumentación del código. Las aplicaciones más populares de esta etapa fueron Wily, Precise o Mercury Interactive, creando las bases de esta técnica, pero fallando a la hora de progresar y fueron barridas en el cambio generacional a las APM 2.0. [96]

Este cambio generacional hacia las APM 2.0 se dio a causa de la popularización de arquitecturas Services Oriented Architecture (SOA), precursora de la arquitectura de microservicios, teniendo como principales diferencias unos servicios menos granulares y con un bus de comunicación centralizado entre ellos. Las APM 2.0 se adaptaron a este nuevo tipo de arquitecturas superando las limitaciones que tenían las APM 1.0 en sistemas distribuidos, ajustándose también a la nueva forma de desarrollar software, con el desarrollo ágil y la integración y entrega continua, en gran parte gracias a la automatización de los mecanismos de instrumentación, pero manteniendo las mismas técnicas de monitorización, es decir, métricas y dashboards [97].

Según Gartner, empresa popular por la calidad de sus análisis de tendencias en el mundo tecnológico, en 2019 definió a las APM como "paquetes de software de monitorización que comprenden la monitorización de experiencia digital (DEM), descubrimiento de aplicaciones, seguimiento y diagnóstico, e inteligencia artificial especialmente diseñada para operaciones de TI" [98]. Esta definición marca un antes y un después en las APM, declarando su alineación casi total a las arquitecturas de microservicios y estableciendo la era de las APM 3.0. Varias de estas características ya se venían implementando en varias APM's, pero esta definición es especialmente significativa por la inclusión del descubrimiento de aplicaciones y la inteligencia artificial como partes fundamentales de las APM's. Otro cambio que pronosticaba el cambio generacional era la creciente adopción de soluciones cloud, contenedores y una gran heterogeneidad en los lenguajes de programación [99]. Las APM's 2.0 más populares como AppDynamics, New Relic y Dynatrace supieron adaptarse e incluso hoy algunas lideran las cotas de adopción y nuevos usuarios. Más adelante trataremos en detalle las herramientas que utilizan las métricas para conseguir observabilidad, por ahora, nos centraremos en establecer las características que constituyen el presente de esta técnica y sus aplicaciones más novedosas, junto con sus buenas prácticas.

Para repasar el estado del arte de esta técnica, primero describiremos el concepto más pequeño con el que se trabaja, la propia métrica y su representación. El conjunto mínimo de métricas para tener una implementación que nos permita ganar observabilidad en un sistema distribuido controlando los aspectos fundamentales de los servicios fue descrito por los ingenieros de Google en 2017, nombrándolo como "Las Cuatro Señales Doradas" [100]. Uno de los mayores aciertos fue definir estas "Señales Doradas" como conceptos deliberadamente amplios y con cierta flexibilidad, dando a los desarrolladores y a los creadores de herramientas que usen esta técnica cierta libertad en sus implementaciones. El

primer tipo son las métricas de latencia, que registran el tiempo que tarda cada servicio en responder. Dentro de la latencia, se recomienda diferenciar las respuestas exitosas de las erróneas, manteniendo un control de ambas, para que los errores no corrompan los tiempos medios de latencia. El segundo tipo son las métricas de tráfico, que medirán la demanda que soporta el sistema medida con una métrica de alto nivel y específica de cada sistema. Por ejemplo, en una aplicación web una aproximación correcta sería con las peticiones HTTP por segundo, en cambio, en un sistema de almacenamiento clave-valor sería más adecuado las transacciones y extracciones de datos por segundo. El tercer tipo de métrica serían los errores, con una base simple que puede extenderse hasta mediciones muy concretas y específicas de los tipos de errores. Miden la tasa de errores en las peticiones, pudiendo estar definidos explícitamente (un error HTTP 500s), por política (tiempo de respuesta superior a un umbral establecido) o implícitamente (cuando la respuesta es HTTP 200 pero no tiene el contenido que se solicitaba). Los errores implícitos surgen para suplir las carencias de los protocolos HTTP y requieren de una capacidad de testeo muy trabajada. Por último, el cuarto tipo de "Señal Dorada" es la saturación, que como mínimo monitorizará la carga de los recursos que se encuentren más demandados en un sistema. Al igual que los errores, se puede ampliar esta monitorización sobre muchos más aspectos, brindando la posibilidad de conocer los % de uso a partir de los cuales los servicios degradan su rendimiento [101]. Estas señales por si solas nos proporcionan una cobertura decente, no obstante, suelen ir acompañadas de otros dos elementos que ayudarán a los desarrolladores a interpretar estos datos, los "dashboards" y las alertas.

Los "dashboards" son paneles en los que podemos controlar todos estos datos, gracias principalmente a la vista de varias gráficas. Ciertas herramientas han pecado de ostentación en este aspecto, a la vez que los usuarios han popularizado herramientas con "dashboards" más sobrios y personalizables. Lo fundamental para disponer de "dashboards" útiles que aprovechen al máximo las métricas recogidas es permitir a los usuarios seleccionar las variables clave de su negocio, así como ofrecer registros históricos y metadatos que proporcionen contexto sobre las métricas que se van incorporando en tiempo real (o lo más próximo posible a este) [102].

Las alertas permiten a los desarrolladores ser conscientes de los incidentes en el funcionamiento de un sistema lo más rápido posible. Sirviéndose de los datos proporcionados por las métricas, es el aspecto que más cambios y evoluciones ha tenido dentro de esta técnica. La IA ha jugado un papel fundamental en esta evolución, ya que su uso para identificar patrones y facilitar el establecimiento de umbrales sobre las diferentes métricas del sistema es un beneficio para los desarrolladores que no podemos pasar por alto. [103]. Además, las alertas se han ido perfeccionando desde simples avisos cuando una métrica sobrepasa cierto umbral hasta llegar a disponer de criterios que, aunque esta sobrepase un umbral, usando el historial del sistema y con el feedback de los desarrolladores sobre verdaderos/falsos positivos/negativos. Estas innovaciones, junto con las mejoras en descriptividad y la filosofía de mantener las alertas al mínimo necesario (promovidas tanto por los creadores de las herramientas como los desarrolladores) [104] son las que, como hemos estudiado anteriormente, llevaron al gran cambio de generación que supuso el salto a las APM 3.0, conformada por las herramientas APM 2.0 pero en sus nuevas y más recientes versiones. En esta generación de herramientas, al igual que en el desarrollo de software, la filosofía Agile se ha ido adoptando también en el uso de las métricas, poniendo énfasis en el proceso de tests y evaluación, mejorando así la efectividad del proceso de monitorización continuo que requieren los sistemas [105]. Los ejemplos más habituales son los tests de carga (evaluando latencia, tráfico y saturación) y los tests unitarios para los errores, cubriendo así las cuatro "Señales Doradas".

No debemos olvidar que además de los aspectos puramente técnicos, la mano y criterio de un equipo es la que resolverá de manera definitiva los incidentes que puedan ocurrir en un sistema, por ello conviene resaltar la importancia de establecer procedimientos de respuesta a incidentes. Desde Netflix, Brendan Gregg, cuyo puesto estaba exclusivamente centrado en el análisis de rendimiento del sistema, impartió una serie de conferencias recalcando la importancia de establecer checklists para localizar y posteriormente resolver un incidente, comparando estas con el proceso de triaje que siguen los pacientes en urgencias hospitalarias [106]. Estas checklists, además de dar un proceso estructurado basado en la experiencia y evolutivo junto con el sistema, proporciona calma en momentos de tensión, por simples que puedan ser. Los pasos varían mucho dentro de cada sistema, pero el proceso más habitual tras conocer el incidente y su impacto suele ser detectando los servicios que presenten problemas de salud o de sus métricas clave, correlacionar sus métricas con otros servicios y repasar el histórico de ellos para dibujar patrones y posibles respuestas a la situación [107].

Antes de analizar las herramientas más populares o con características más novedosas, trataremos los aspectos más importantes que sus arquitecturas deben proporcionar, según un estudio realizado por Instana a la comunidad [108]. Entre otras conclusiones, destacan 7 puntos principales para aplicar esta técnica en un sistema distribuido dinámico y escalable:

- Disponibilidad de datos en tiempo real o cuasi-real
- Descubrimiento automático y continuo de la topología del sistema para conseguir información "semántica" necesaria para correlacionar datos y analizar la salud general.
- Persistencia histórica de los servicios y componentes para estudiar la evolución del sistema.
- Actualización continua del estado y seguimiento de todos los cambios en los componentes para brindar un servicio estable.
- Definición de las métricas clave de los servicios del sistema para su análisis y poder extraer el gran potencial de esta técnica.
- Recopilación automática y análisis de métricas que facilite entender el estado de los componentes y servicios, localizando las causas raíz de los incidentes.
- Introducción de datos definidos por los desarrolladores, integrándolos en la semántica e inteligencia de la herramienta, para facilitar la automatización de monitorización y alerta automáticas.

Las herramientas que recogen métricas para ganar observabilidad han sufrido una gran evolución, como hemos podido ver anteriormente, pero debido a su actual nexo con los otros dos pilares de la observabilidad repasaremos herramientas centradas exclusivamente en las métricas y las novedades que han aportado a esta técnica.

Como primera herramienta analizaremos StatsD. Fue de las primeras en ofrecer métricas en tiempo real desde su nacimiento en 2011, funcionando como un demonio (también llamado StatsD Server) que recibe las métricas mediante UDP, recogidas de los servicios previamente instrumentados con código gracias a la multitud de librerías que ofrece, y las agrega. Tras esto, el servidor las envía a intervalos regulares a un backend que se encargará de almacenarlas y/o realizar los análisis o transformaciones convenientes [109]. Es una herramienta OpenSource y permite intercambiar cualquiera de sus partes sin afectar a las demás, dando una gran flexibilidad a los equipos. La estructura de las métricas es sencilla y eficaz debido a los tipos diferentes que ofrece (permitiéndonos capturar las "Señales Doradas"), componiéndose de nombre, valor y tipo. Los tipos de serie que ofrece StatsD son contadores, muestreos, gauges (un contador que además de incrementar puede

decrecer), temporizadores o conjuntos de estas sobre una métrica del código del servicio que queramos registrar (previamente instrumentada), pudiendo ampliar estos tipos gracias a sus librerías y añadir por ejemplo métricas sobre uso de recursos hardware en un sistema Linux [110]. En cuanto a los backends a los que podemos conectar al demonio, poseemos una gran variedad de implementaciones, desde herramientas que simplemente almacenen los datos de las métricas en TSDB's hasta que permitan mostrar gráficos, alertas y análisis de correlación de eventos. Entre ellas, destaca DataDog, que impulsó una de las últimas innovaciones de StatsD, el uso de agentes para facilitar la colección de métricas eliminando la necesidad del servidor mandando directamente los datos a este backend. Además, añadieron la capacidad de etiquetar las métricas, pudiendo filtrar y correlacionar bajo parámetros que puedan compartirse entre servicios (por ejemplo, para discernir entre los tipos de peticiones HTTP), posibilidad de la que por defecto StatsD carece [111]. A pesar de su aparente complejidad debido a su arquitectura, StatsD triunfó gracias a su verdadera simplicidad de uso y el enorme esfuerzo de su comunidad por desarrollar librerías que permitiesen implementaciones en multitud de lenguajes (gracias a su carácter OpenSource). Otro beneficio que aportó fue desacoplar la aplicación de su instrumentación, haciendo que, si StatsD cae, no afecte al funcionamiento de esta. Esta herramienta tiene como lema "medir todo" y su popularización permeó en esta técnica hasta llegar a conformar un estándar en la industria del software para monitorizar e instrumentar aplicaciones [112] y que otras herramientas desarrollasen sus integraciones con este protocolo.

Con la popularización de soluciones en Cloud, en concreto Kubernetes, tanto el diseño de StatsD como su modelo de datos quedaron relegados a un segundo plano (que posteriormente sería resuelto por herramientas como DataDog o NewRelic con sus integraciones). Ante esta situación, los ingenieros de SoundCloud en 2012 desarrollaron una herramienta que supliese estas carencias, proporcionando un modelo de datos multi-dimensional para poder diferenciar servicio, instancia, endpoint y método, simplificando la operación del sistema desacoplando parte de la configuración, mejorando y facilitando la capacidad de escalado y creando un lenguaje de consultas que aproveche el nuevo modelo de datos para generar alertas informativas y gráficas [113]. Esta herramienta es Prometheus, comenzó en 2012 como un proyecto OpenSource, pero hasta 2015 no se lanzó de manera oficial, ya que los ingenieros de SoundCloud querían disponer de flexibilidad para hacer cambios y darle suficiente madurez al proyecto. En tan solo un año, alcanzó una enorme popularidad, llegando a ser el 4º repositorio en GitHub en tendencias [114].

El modelo de datos de Prometheus corrige las debilidades de StatsD, proporcionando una estructura más flexible y potente, constando únicamente de dos partes, el nombre de la métrica y un conjunto de claves-valor que serán las etiquetas, dándonos esa potencial dimensionalidad que mencionábamos anteriormente. Estas métricas serán almacenadas como series temporales en TSDB's, creando una nueva serie temporal por cada cambio en las etiquetas. Además, Prometheus actualiza los tipos de métricas a contadores, gauges (un contador que además de incrementar puede decrecer), histogramas y resúmenes. Estos dos últimos presentan una complejidad mucho mayor que lo visto anteriormente, siendo los histogramas muestreos de observaciones de una métrica contados en "buckets" configurables. Los resúmenes actúan de manera similar, pero calculan cuantiles previamente configurados, y son aconsejados para mediciones precisas de cuantiles. Además de todas estas innovaciones orientadas a permitir una obtención mucho más diversa de métricas, una característica que impulsó la adopción de esta herramienta fue la generación automática de métricas para las instancias de servicios, permitiendo de serie su adopción en sistemas distribuidos y con una alta escalabilidad [115].

Para aprovechar el potencial de estas métricas, Prometheus diseñó un nuevo lenguaje de consultas muy potente, PromQL, que permite "cortar y rebanar" los datos a placer, crear

gráficos, visualización de tablas en navegador, o servir datos a sistemas externos a través de la API HTTP. Los tipos de datos sobre los que trabaja son escalares, vectores instantáneos y vectores de rango y su uso depende de la consulta que queramos hacer ya que para ciertos casos solo algunos de esos tipos están permitidos. Los escalares son simples números de coma flotante, los vectores instantáneos son conjuntos de escalares en un punto en el tiempo y los vectores de rango son un conjunto de vectores insantáneos, aunque estos últimos no pueden ser usados directamente por los gráficos, pero son muy útiles porque facilitan el uso las funciones que nos ofrece PromQL [116]. Para hacer los análisis y consultas que necesitemos, nos ofrece operadores del tipo aritmético, lógico u operadores para comparar vectores y encontrar coincidencias 1 a 1, N a 1 o 1 a N, además de una gran cantidad de operadores para agregar datos para hacer análisis exhaustivos, permitiéndonos hacer esas "preguntas" que necesita la observabilidad [117]. Por último, en cuanto al modelo de datos de Prometheus, las funciones que nos ofrece son similares a un lenguaje de programación. Nos permiten de manera sencilla operar con los datos, pudiendo encontrar funciones para redondear, hacer cálculos o manipular las etiquetas, teniendo como inconveniente que la mayoría de ellas son aproximaciones y extrapolan los resultados [118]. Este inconveniente por sí solo puede hacernos descartar Prometheus como herramienta en situaciones en las que necesitemos una gran exactitud.

En cuanto a la arquitectura de Prometheus, esta aportó una serie de novedades que hoy en día siguen sosteniéndola como una de las herramientas más populares para aplicar esta técnica y llegar a ser reconocida como el tercer proyecto graduado en la CNCF [119]. Su instrumentación es similar a la de StatsD, permitiéndonos hacer métricas personalizadas y dado que disponemos de etiquetas el potencial y la flexibilidad son mayores. Una de las innovaciones que aportó fue que la instrumentación de endpoints fuese mucho más sencilla y potente, pudiendo instrumentarlos desde configuración. Los servicios instrumentados generarán métricas que posteriormente serán recogidas periódicamente por el Prometheus Server a través de sus endpoints. Esto es posible gracias al descubrimiento de servicios del que dispone Prometheus, el cuál puede configurarse a través de multitud de mecanismos [120], y supuso un salto adelante en la adopción de soluciones en la nube. Además, el servidor almacenará las métricas y se comunicará con Alertmanager y a través de PromQL con su interfaz web u otra herramienta preparada para integrarse con Prometheus, como Grafana. El Alertmanager es otra pieza bastante novedosa, que desacopla la configuración de las alertas y nos permite agruparlas, silenciarlas, inhibirlas o enrutarlas a otras herramientas, incluso al correo electrónico [121]. Todas estas características son las que convirtieron a Prometheus en la herramienta de métricas más popular y su constante evolución y modernización, como indica su reciente adopción del uso de agentes, permitiendo una mejor escalabilidad horizontal, además de ser un proyecto de la CNCF, promete un futuro brillante [122].

Habiendo repasado estas herramientas y su determinante influencia en las métricas como técnica para ganar observabilidad, es momento de abordar el elefante en la habitación de las métricas. El uso de TSDB's era un estándar de facto hasta hace muy poco en esta técnica y, como anteriormente hemos descrito, presenta un problema en sistemas que generen métricas con alta cardinalidad. Este problema aumenta si consideramos que los sistemas distribuidos aumentan las variables que debemos medir con esta técnica, agravando este problema.

Ciertas de las anteriores herramientas presentan carencias en este aspecto, recomendando simplemente controlar el número de series temporales activas y centrarse en monitorizar variables realmente valiosas, supliendo los puntos débiles de no disponer de alta cardinalidad con otras herramientas [92]. En los últimos años, han surgido propuestas que intentan solventar esta problemática con enfoques relativamente diferentes y originales.

Uno de los más recientes es el de Grafana Mimir, una TSDB para Prometheus que, entre otras optimizaciones, usa réplicas y particiones junto con un esquema de hasheo consistente llamado Hash Ring, con el que consiguieron mantener mil millones de series activas (que recibieron una métrica hace <20 min) con un éxito del 99,9% de escrituras y lecturas, siendo estas menores de 10 segundos y 2 segundos de media, respectivamente [123]. Este Hash Ring está enfocado en aprovechar la partición de datos y evitar la degradación a partir de cierto umbral que sufren las TSDB's, y no es la única aproximación de este estilo, ya que otras comunidades y herramientas han optado por soluciones similares. Entre ellas, encontramos a InfluxDB con su "Time Series Index", que actúa como una base de datos basada en "log-structured merge-tree" con complejidad de O(N) en lectura y escritura [124], que al igual que Grafana Mimir, está diseñado para beneficiarse de la partición de datos [125]. Otro caso similar es el de TimescaleDB y su uso de los árboles B para particionar la base de datos a lo largo del tiempo, dando al usuario la apariencia de estar manejando una sola "hipertabla" cuando está manejando una base de datos distribuida y manejada por una o varias instancias de TimescaleDB. [126]. Una aproximación diferente que mejora la capacidad de computación es el uso de codificación en los datos (que en algunos casos ha ahorrado el 90% de los costes en almacenamiento), entre ellos siendo muy eficaz para números enteros la codificación delta y delta-delta, que almacena la diferencia entre un objeto y un objeto de referencia. En cuanto a números de coma flotante, la compresión XOR ha presentado reducciones de hasta 12 veces el tamaño original de una serie temporal [127]. En cuanto a las APM's 3.0, su mayor representante es DataDog, aunque también son dignos de mención los esfuerzos innovadores de Dynatrace y New Relic. Esta generación se encuentra en medio de esta transición, solucionando el problema que representa el uso de las TSDB's junto con alta cardinalidad, a pesar de haberse adaptado con bastante éxito a los microservicios e incluso adoptado OpenTelemetry. En este apartado destaca Signoz, una APM muy novedosa y reciente, que incorpora tanto eventos estructurados como bases de datos columnares, que trataremos más adelante y comprenderemos la revolución que suponen. [128]

Las métricas son una técnica que, aunque requiera de cierto trabajo previo y esfuerzo por parte de los desarrolladores, presenta una mejora notable en la observabilidad de los sistemas, pudiendo controlar el funcionamiento y rendimiento desde puntos sensibles a sistemas completos. En los últimos años, como hemos descrito en los anteriores párrafos, ha sufrido una evolución vertiginosa debido al entorno distribuido que cada vez impera más en el mundo del software, y tanto iniciativas privadas como la comunidad Open Source han presentado enfoques y mejoras muy notables que cimentan los caminos para el futuro de esta técnica. Aun con todos estos esfuerzos, es una técnica que necesita de otros apoyos para conseguir esa deseada visión holística del sistema, por ello una gran cantidad de APM's que hemos mencionado también incorporan el uso de logs y trazabilidad distribuida para complementar a las métricas, además de soportar de manera generalizada OpenTelemetry, llegando a desdibujar sus limitaciones y acercándose a la observabilidad.

CONCLUSIONES

La industria del software cambió de forma radical con la aparición de la computación en la nube, y el mundo todavía sigue adaptándose a los efectos que esto trajo consigo. Junto a esta nueva y potente posibilidad para los desarrolladores y equipos, surgieron la arquitectura de microservicios y las prácticas de entrega continua y despliegue continuo (CI/CD).

Esta nueva concepción sobre cómo se construye, despliega y mantiene el software alentó el debate sobre el tema principal de este trabajo, la observabilidad en los sistemas distribuidos. Esta técnica y sus tres principales "pilares" tenían recorrido, aunque limitado, antes de la aparición de esta nueva concepción. Como hemos descrito anteriormente, los entornos emergentemente caóticos que son los sistemas distribuidos, teniendo como representante principal a los microservicios, generan una gran incertidumbre sobre su funcionamiento debido a la inherente complejidad de su arquitectura.

La observabilidad en el mundo del software se propuso como solución para atajar esta falta de conocimiento de los desarrolladores y equipos sobre sus propios sistemas, y su adopción y desarrollo ha sido exponencial, ganando nuevos usuarios cada año de forma constante [129], además de las innovaciones constantes que han ido presentando las herramientas que aplican esta técnica, como hemos repasado en el estado del arte de los llamados "tres pilares de la observabilidad".

Llegando al presente y habiendo comprendido el trasfondo histórico de esta técnica, se nos presenta un futuro prometedor sin ninguna duda, donde los usuarios puedan reducir al mínimo su desconocimiento sobre sus sistemas y los tiempos de respuesta en caso de incidentes. Las tres técnicas que conforman estos pilares presentan debilidades estructurales por si solas y no pueden otorgar observabilidad completa, debido a que pierden esa capacidad de correlacionar datos de diferentes niveles para poder extraer conclusiones. Centrándonos en cada una de las técnicas, podemos identificar los principales puntos que presentan mayores dificultades o problemas a los equipos que las implementan.

En primer lugar, la trazabilidad distribuida es una técnica muy robusta y con un concepto teórico que encaja idealmente con estas arquitecturas, manteniendo su rendimiento en supuestos de escalado gracias a su naturaleza distribuida. En caso de mencionar una debilidad actual de esta técnica, y prácticamente la única a la que se enfrenta, es que sólo nos proporciona datos de alto nivel y poco granulares respecto a lo que ocurre internamente en un servicio, no obstante, siendo tremendamente reveladores respecto a la visión panorámica del sistema [130]. Esta carencia no se acerca a las mayores preocupaciones de los desarrolladores de sistemas que utilizan esta técnica ni de los que equipos que crean herramientas basadas en ella, ya que asumen que la observabilidad no es algo que se pueda conseguir con una sola técnica. Actualmente centran sus demandas y esfuerzos en estandarizar esta técnica para superar las ataduras a un vendedor específico, con el apoyo de organizaciones tan importantes como la CNCF [131]. Por estos motivos, la trazabilidad distribuida se sitúa en una etapa de consolidación que aumentará su adopción ofreciendo comodidad y flexibilidad a los desarrolladores que decidan utilizar esta técnica para ganar observabilidad en sus sistemas.

En segundo lugar, los logs son la técnica con mayor recorrido de los tres pilares, pero no por ello es la "mejor" de las tres, sino que las otras técnicas surgieron para suplir carencias de esta o necesidades específicas. El punto fuerte de esta técnica es que ataca el punto débil de la trazabilidad distribuida, permitiéndonos recoger datos de alta granularidad de los servicios para conocer con exactitud datos sobre el funcionamiento de los servicios. En

cambio, esta técnica presenta ciertos retos en su implementación y mantenimiento debido a la constante generación de logs y la dificultad inherente que tiene realizar búsquedas entre ellos [132]. Es por esto que el futuro de esta técnica no depende tanto de nuevos avances en cuanto a recolección y/o mejoras en las herramientas, sino de un buen uso de las buenas prácticas expuestas por los equipos, en concreto la estructuración, el uso de logs en puntos críticos y su etiquetado según procedencia [133].

Por último, las métricas se encuentran en una posición delicada. A pesar de los constantes avances y mejoras espectaculares en cuanto a rendimiento y precisión, como por ejemplo el caso de Grafana Mimir, se asientan sobre una base teórica que nunca podrá alinearse con la observabilidad de un sistema distribuido. Esta técnica agrega datos recogidos en instantes de tiempo, si bien para monitorizar recursos hardware es una aproximación completamente válida, en cuanto al funcionamiento del sistema perder parte del contexto puede resultar en ignorar errores y no poder extraer conclusiones para resolver incidentes, y al agregar datos de diferentes servicios nos podemos encontrar en esa situación [134]. A pesar de esta desalineación entre ambos conceptos, las métricas siguen jugando un papel muy importante en este asunto debido a que en la gran mayoría de los casos el uso de esta técnica nos permite extraer conclusiones muy reveladoras sobre el funcionamiento de un sistema. Además, continúa recibiendo el apoyo de la CNCF para su estandarización con OpenMetrics (muy similar al caso de OpenTelemetry) y las herramientas que aplican esta técnica, junto con las APM, siguen creciendo año tras año en adopción [135].

El debate sobre las métricas se ha intensificado recientemente, y en general sobre el concepto de observabilidad, que ya de por sí nació como un concepto polémico. Es en este punto donde se juega el futuro de la observabilidad, entre las tendencias más continuistas y propuestas muy rupturistas e innovadoras. Hasta ahora, hemos descrito las primeras, pero es muy interesante que exploremos las propuestas más rompedoras con los conceptos que fundamentan la observabilidad para poder comprender los posibles caminos futuros de esta.

Destaca sobre todo la propuesta de los eventos estructurados y las bases de datos columnares, por su gran diferencia con la situación actual de la observabilidad. Los eventos estructurados son un formato de datos en los que se guardan parejas de clave-valor con un número arbitrario de campos o dimensiones. Pueden contener datos relativamente simples y relativos al servicio hasta ser extendidos con id's de procesos "padres" [136]. El enfoque principal de esta técnica, más que sustituir a los actuales pilares, es reemplazar a las métricas, o, en todo caso, actuar como un cuarto pilar debido a lo novedosa que supone esta propuesta junto con su uso de bases de datos columnares. Estas bases de datos no necesitan crear una serie temporal nueva por cada métrica diferente, sino que permiten la creación de columnas nuevas por cada tipo de métrica nuevo, siendo mucho más flexibles y atajando finalmente el problema de la alta cardinalidad en las TSDB's [137].

Aunque resulte atractivo teorizar sobre si los eventos estructurados han llegado para acabar con las métricas, la realidad es mucho más compleja que un debate tan corto de miras. El 72% de equipos que están preocupados por la observabilidad en sus sistemas actualmente utilizan dos o más herramientas [138], pudiendo utilizar varias que se solapan en sus propósitos. Siendo conscientes de la situación y de las capacidades actuales de las técnicas que integran la observabilidad, muy probablemente el camino a corto plazo estará marcado por la normalización y unificación de datos, que nos permita saltar entre los tres pilares de manera fluida, manipulándolos y correlacionándolos, aunque tengan diferentes procedencias o estructuras. Este camino lo lideran la CNCF y herramientas como Dynatrace o Datadog, siendo estas las dos últimas las que encabezan el cuadrante de la observabilidad de Gartner, en el que además aparece Honeycomb, la primera herramienta

que utiliza los eventos estructurados en conjunto con los tres pilares [139]. El principal nexo entre estas herramientas es el tratamiento conjunto que ofrecen de las diferentes técnicas, consiguiendo la deseada capacidad de formular preguntas sin estar limitado por el alcance de las técnicas que integran la observabilidad.

En su poco tiempo de vida, no alcanzando prácticamente ni el quinquenio desde el inicio de los debates sobre su base teórica, la observabilidad se ha establecido como una de grandes las prioridades que un equipo tiene que valorar a la hora de diseñar una arquitectura basada en microservicios o migrar su sistema a esta arquitectura. Esto no es de extrañar debido a los beneficios que aporta en aspectos tan variados e importantes como lo son la reducción de tiempos de respuesta a incidentes, seguridad y tranquilidad de los desarrolladores sabiendo que no tienen que revisar uno por uno los logs de sus decenas de servicios o poder reflexionar sobre el funcionamiento del sistema y mejorar sus partes débiles.

A pesar de todos los beneficios que ofrece la observabilidad, las organizaciones suelen ofrecer resistencia a los cambios de manera natural, y no es hasta situaciones como, por ejemplo, una caída catastrófica del servicio, en la que empiezan a sopesar sus carencias en este aspecto. Conseguir un cambio de paradigma a nivel socio-técnico en las organizaciones que utilicen microservicios es complicado, al igual que lo fue y sigue siendo la transición del monolito a esta arquitectura. Por ello estimamos necesario destacar esfuerzo divulgativo y de investigación que se realiza desde la comunidad, tanto los artículos que aparecen referenciados como muchos otros que no lo están, pero igualmente ayudan a promover este avance tan necesario y, en especial, destacar el papel de Charity Majors (CTO Honeycomb, herramienta ya mencionada por su innovadora aproximación con eventos estructurados) y colaboradores por su publicación del libro "Observability Engineering", en el que se trata este tema a un nivel mucho más práctico pero con un gran trasfondo teórico, marcando un punto de inflexión que seguramente propiciará una mayor adopción de esta técnica. [140]

Aun enfrentándose a estos retos, la observabilidad muy probablemente crecerá en importancia, y su camino, independientemente de si las APM se renombran, si los eventos estructurados sustituyen a las métricas o se integran como un cuarto pilar, no se desviará de ofrecer a desarrolladores, equipos y clientes las ventajas que hemos descrito, con soluciones cada vez más flexibles que se puedan ajustar a la miríada de necesidades concretas que tengan de un sistema que implementa la arquitectura de microservicios.

BIBLIOGRAFÍA

[1] M. Loukides y S. Swoyer, «O'Reilly,» 15 Julio 2020. [En línea]. Available: https://www.oreilly.com/radar/microservices-adoption-in-2020/. [Último acceso: Junio 2022].

[2] «Charter Global,» [En línea]. Available: https://www.charterglobal.com/five-microservices-trends-in-2020/. [Último acceso: Junio 2022].

[3] M. Fowler y J. Lewis, 25 Marzo 2014. [En línea]. Available: https://martinfowler.com/articles/microservices.html. [Último acceso: Junio 2022].

[4] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2003.

[5] R. C. Martin, «Cleancoder,» 8 Mayo 2014. [En línea]. Available: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. [Último acceso: Junio 2022].

[6] M. Fowler, «MartinFowler,» 21 Agosto 2019. [En línea]. Available: https://www.martinfowler.com/microservices/. [Último acceso: Junio 2022].

[7] R. Awati y I. Wigmore , «TechTarget,» [En línea]. Available: https://www.techtarget.com/whatis/definition/monolithic-architecture. [Último acceso: Junio 2022].

[8] IBM Cloud Team, «IBM,» 14 Mayo 2021. [En línea]. Available: https://www.ibm.com/cloud/blog/soa-vs-microservices. [Último acceso: Junio 2022].

[9] A. Mauricio, «Ouracademy,» 17 Febrero 2020. [En línea]. Available: https://our-academy.org/posts/microservicios-que-son-cuando-usarlos-y-como-construirlos. [Último acceso: Junio 2022].

[10] Amazon Web Services, «AWS,» 5 Agosto 2021. [En línea]. Available: https://docs.aws.amazon.com/whitepapers/latest/running-containerized-microservices/welcome.html. [Último acceso: Junio 2022].

[11] C. Richardson, «Microservices,» 2021. [En línea]. Available: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html. [Último acceso: Junio 2022].

[12] M. E. Conway, «MelConway,» Abril 1968. [En línea]. Available: http://www.melconway.com/Home/Committees_Paper.html. [Último acceso: Junio 2022].

[13] S. Narayan, «MartinFowler,» 20 Febrero 2018. [En línea]. Available: https://martinfowler.com/articles/products-over-projects.html. [Último acceso: Junio 2022].

[14] S. Behara , 16 Mayo 2019. [En línea]. Available: https://samirbehara.com/2019/05/16/eight-fallacies-of-distributed-computing/. [Último acceso: Junio 2022].

[15] A. Kamil, «Medium,» 3 Marzo 2019. [En línea]. Available: https://medium.com/@kamilmasyhur/rabbitmq-what-is-that-10c74ac7620a. [Último acceso: Junio 2022].

[16] N. Peck, «Medium,» 5 Septiembre 2017. [En línea]. Available: https://medium.com/@nathankpeck/microservice-principles-decentralized-governance-4cdbde2ff6ca. [Último acceso: Junio 2022].

[17] S. Viatchanin, «MadDevs,» 31 Marzo 2021. [En línea]. Available: https://maddevs.io/blog/tech-stack-of-prominent-companies/. [Último acceso: Junio 2022].

[18] N. Peck, «Medium,» 14 Septiembre 2017. [En línea]. Available: https://medium.com/@nathankpeck/microservice-principles-decentralized-data-management-4adaceea173f. [Último acceso: Junio 2022].

[19] «VMware,» [En línea]. Available: https://www.vmware.com/topics/glossary/content/infrastructure-automation.html. [Último acceso: Junio 2022].

[20] Amazon Web Services, «AWS,» 5 Agosto 2021. [En línea]. Available: https://docs.aws.amazon.com/whitepapers/latest/running-containerized-microservices/design-for-failure.html. [Último acceso: Junio 2022].

[21] L. Cameron, «IEEE Computer Society,» 17 Septiembre 2018. [En línea]. Available: https://www.computer.org/publications/tech-news/research/realizing-software-reliability-in-the-face-of-infrastructure-instability. [Último acceso: Junio 2022].

[22] P. Debois, «Agile Infrastructure and Operations: How Infra-gile are You?,» de *Agile 2008 Conference*, Toronto, ON, Canada , 2008.

[23] C. Tozzi, «DigitalisationWord,» Mayo 2021. [En línea]. Available: https://m.digitalisationworld.com/blogs/56635/observability-vs-monitoring-whats-the-difference. [Último acceso: Junio 2022].

[24] R. E. Kalman, «On the general theory of control systems,» *IFAC Proceedings Volumes,* vol. 1, nº 1, pp. 491-502, 1960.

[25] B. Cantrill, «Visualizing Distributed Systems with Statemaps,» de *KubeCon*, Seattle, Washington, USA, 2018.

[26] IEEE, «IEEE Standard Glossary of Software Engineering Terminology,» *IEEE,* pp. 1-84, 1990.

[27] C. Sridharan, Distributed Systems Observability, O'Reilly Media, Inc., 2018 Julio.

[28] T. Treat, «BraveNewGeek,» 3 Octubre 2019. [En línea]. Available: https://bravenewgeek.com/microservice-observability-part-1-disambiguating-observability-and-monitoring/. [Último acceso: Junio 2022].

[29] Y. Shkuro, «Medium,» 17 Mayo 2019. [En línea]. Available: https://medium.com/@YuriShkuro/observability-challenges-in-microservices-and-cloud-native-applications-72857f9d03af. [Último acceso: Junio 2022].

[30] M. Gomez, «InfoQ,» 15 Agosto 2019. [En línea]. Available:

https://www.infoq.com/presentations/monolith-observable-microservices-ddd/. [Último acceso: Junio 2022].

[31] «Observe,» 16 Febrero 2021. [En línea]. Available: https://www.observeinc.com/resources/what-is-observability/. [Último acceso: Junio 2022].

[32] S. Niedermaier, F. Koetter, A. Freymann y S. Wagner , «On Observability and Monitoring of Distributed Systems – An Industry Interview Study,» de *International Conference on Service-Oriented Computing*, Toulouse, Francia, 2019.

[33] C. Klein, «CNCF,» 2 Marzo 2021. [En línea]. Available: https://www.cncf.io/blog/2021/03/02/what-was-observability-again/. [Último acceso: Junio 2022].

[34] J. Livens, «Dynatrace,» 27 Septiembre 2021. [En línea]. Available: https://www.dynatrace.com/news/blog/what-is-distributed-tracing/. [Último acceso: Julio 2022].

[35] Souvik, «Deepsource,» 14 Abril 2020. [En línea]. Available: https://deepsource.io/blog/distributed-tracing/. [Último acceso: Julio 2022].

[36] [En línea]. Available: https://www.splunk.com/en_us/data-insider/what-is-distributed-tracing.html#distributed-tracing. [Último acceso: Julio 2022].

[37] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan y C. Shanbhag, «Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,» Google Inc., 2010.

[38] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds y A. Muthitacha, «Performance debugging for distributed systems of black boxes,» *Operating Systems Review,* vol. 37, nº 5, p. 74–89, 2003.

[39] P. Barham, R. Isaacs, R. Mortier y D. Narayanan, «Magpie: Online Modelling and Performance-aware Systems,» *USENIX,* 2003.

[40] ASF Infrabot, «Apache,» 10 Abril 2019. [En línea]. Available: https://cwiki.apache.org/confluence/display/incubator/ZipkinProposal. [Último acceso: Julio 2022].

[41] «Zipkin,» [En línea]. Available: https://zipkin.io/pages/architecture.html. [Último acceso: Julio 2022].

[42] «Zipkin,» [En línea]. Available: https://zipkin.io/pages/tracers_instrumentation.html. [Último acceso: Julio 2022].

[43] «Zipkin,» [En línea]. Available: https://zipkin.io/zipkin-api/#/default/post_spans. [Último acceso: Julio 2022].

[44] SentinelOne , «SentinelOne,» 6 Enero 2021. [En línea]. Available: https://www.sentinelone.com/blog/zipkin-tutorial-distributed-tracing/. [Último acceso: Julio 2022].

[45] J. Stein, 12 Febrero 2016. [En línea]. Available: https://groups.google.com/g/zipkin-user/c/v-Yw6G8W0kg. [Último acceso: Julio 2022].

[46] K. Chaturvedi, «Medium,» 12 Julio 2019. [En línea]. Available: https://medium.com/modern-distributed-tracing-systems-architecture/modern-distributed-tracing-systems-architecture-698691b4f998. [Último acceso: Julio 2022].

[47] openzipkin, «Github,» [En línea]. Available: https://github.com/openzipkin/zipkin. [Último acceso: Junio 2022].

[48] G. Arbezzano, «TheNewStack,» 10 Octubre 2017. [En línea]. Available: https://thenewstack.io/opentracing-open-standard-distributed-tracing/. [Último acceso: Julio 2022].

[49] OpenTracing, «OpenTracing,» [En línea]. Available: https://opentracing.io/docs/overview/what-is-tracing/#what-is-opentracing. [Último acceso: Julio 2022].

[50] N. Šabić, «Sematext,» 2 Marzo 2019. [En línea]. Available: https://sematext.com/blog/how-opentracing-works/. [Último acceso: Julio 2022].

[51] OpenTracing, «Github,» 13 Agosto 2021. [En línea]. Available: https://github.com/opentracing/specification/blob/master/specification.md. [Último acceso: Julio 2022].

[52] Y. Shkuro, «Uber,» 2 Febrero 2017. [En línea]. Available: https://eng.uber.com/distributed-tracing/. [Último acceso: Julio 2022].

[53] L. Mora, «ParadigmaDigital,» 26 Noviembre 2018. [En línea]. Available: https://www.paradigmadigital.com/dev/trazabilidad-distribuida-con-opentracing-y-jaeger/. [Último acceso: Julio 2022].

[54] Jaeger, «Openshift,» [En línea]. Available: https://docs.openshift.com/container-platform/4.5/jaeger/jaeger_arch/rhbjaeger-architecture.html. [Último acceso: Julio 2022].

[55] Jaeger, «JaegerTracing,» 5 Julio 2022. [En línea]. Available: https://www.jaegertracing.io/docs/1.36/architecture/. [Último acceso: Julio 2022].

[56] J. Louwers, «Medium,» 5 Enero 2021. [En línea]. Available: https://louwersj.medium.com/using-jaeger-collector-with-kafka-2d5473c655a5. [Último acceso: Julio 2022].

[57] CNCF, «CNCF,» 13 Septiembre 2017. [En línea]. Available: https://www.cncf.io/projects/jaeger/. [Último acceso: Julio 2022].

[58] S. Özal , «TheNewStack,» 15 Octubre 2020. [En línea]. Available: https://thenewstack.io/jaeger-vs-zipkin-battle-of-the-open-source-tracing-tools/. [Último acceso: Julio 2022].

[59] «OpenCensus,» [En línea]. Available: https://opencensus.io/. [Último acceso: Julio 2022].

[60] B. Sigelman y M. McLean, «CNCF,» 21 Mayo 2019. [En línea]. Available: https://www.cncf.io/blog/2019/05/21/a-brief-history-of-opentelemetry-so-far/. [Último acceso: Julio 2022].

[61] T. Young, «Medium,» 19 Abril 2019. [En línea]. Available:

https://medium.com/opentracing/a-roadmap-to-convergence-b074e5815289. [Último acceso: Julio 2022].

[62] CNCF, «CNCF,» 26 Agosto 2021. [En línea]. Available: https://www.cncf.io/blog/2021/08/26/opentelemetry-becomes-a-cncf-incubating-project/. [Último acceso: Julio 2022].

[63] OpenTelemetry, «OpenTelemetry,» 10 Junio 2022. [En línea]. Available: https://opentelemetry.io/docs/concepts/. [Último acceso: Julio 2022].

[64] P. Loffay, «Medium,» 27 Junio 2021. [En línea]. Available: https://ploffay.medium.com/five-years-evolution-of-open-source-distributed-tracing-ec1c5a5dd1ac. [Último acceso: Julio 2022].

[65] Humio, «Humio,» 22 Septiembre 2021. [En línea]. Available: https://www.humio.com/glossary/log-file/. [Último acceso: Julio 2022].

[66] D. Lee, «XPLG,» 24 Mayo 2020. [En línea]. Available: https://www.xplg.com/application-logs-what-how/. [Último acceso: Julio 2022].

[67] «Wikipedia,» 26 Junio 2022. [En línea]. Available: https://en.wikipedia.org/wiki/Java_version_history. [Último acceso: Julio 2022].

[68] C. Eberhardt, «CodeProject,» 24 Marzo 2014. [En línea]. Available: https://www.codeproject.com/Articles/42354/The-Art-of-Logging. [Último acceso: Julio 2022].

[69] M. Michels, «Maximilian Michels Blog,» 7 Octubre 2020. [En línea]. Available: https://maximilianmichels.com/2020/debugging-distributed-systems/. [Último acceso: Julio 2022].

[70] R. Ribenzaft, «Epsagon,» 6 Junio 2019. [En línea]. Available: https://epsagon.com/observability/5-ways-to-understand-distributed-system-logging-and-monitoring/. [Último acceso: Julio 2022].

[71] J. Kanjilal , «TechTarget,» 6 Noviembre 2020. [En línea]. Available: https://www.techtarget.com/searchapparchitecture/tip/5-essential-tips-for-logging-microservices. [Último acceso: Julio 2022].

[72] GrayLog, «GrayLog,» 17 Enero 2019. [En línea]. Available: https://www.graylog.org/post/log-file-parsing. [Último acceso: Julio 2022].

[73] SentinelOne, «DataSet,» 15 Octubre 2019. [En línea]. Available: https://www.dataset.com/blog/the-10-commandments-of-logging/. [Último acceso: Julio 2022].

[74] C. Richardson, «Microservices,» [En línea]. Available: https://microservices.io/patterns/observability/application-logging.html. [Último acceso: Julio 2022].

[75] G. T. Davis, «Treasure Data,» 6 Diciembre 2019. [En línea]. Available: https://blog.treasuredata.com/blog/2016/08/03/distributed-logging-architecture-in-the-container-era/. [Último acceso: Julio 2022].

[76] G. T. Davis, «DZone,» 10 Agosto 2016. [En línea]. Available: https://dzone.com/articles/distributed-logging-in-the-container-era-part-2-1. [Último

acceso: Julio 2022].

[77] S. P. R. Janapati , «DZone,» 1 Agosto 2017. [En línea]. Available: https://dzone.com/articles/distributed-logging-architecture-for-microservices. [Último acceso: Julio 2022].

[78] P. He, «An End-To-End Log Management Framework for Distributed Systems,» de *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, Hong Kong, China, 2017.

[79] P. He, J. Zhu, Z. Zheng y M. R. Ly, «Drain: An Online Log Parsing Approach with Fixed Depth Tree,» de *2017 IEEE International Conference on Web Services (ICWS)*, Honolulu, Hawaii, USA, 2017.

[80] Z. Chen, J. Liu, W. Gu, Y. Su y M. R. Lyu, «Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection,» arXiv, 2021.

[81] GrayLog, «GrayLog,» 28 Febrero 2022. [En línea]. Available: https://docs.graylog.org/docs. [Último acceso: Julio 2022].

[82] D. Horovits, «Logz,» 9 Marzo 2022. [En línea]. Available: https://logz.io/blog/the-cost-of-doing-elk-stack-on-your-own/. [Último acceso: Julio 2022].

[83] P. Pedamkar, «Educba,» [En línea]. Available: https://www.educba.com/graylog-vs-elk/. [Último acceso: Julio 2022].

[84] CNCF, «CNCF,» 8 Noviembre 2016. [En línea]. Available: https://www.cncf.io/projects/fluentd/. [Último acceso: Julio 2022].

[85] R. Ribenzaft, «CNCF,» 26 Febrero 2020. [En línea]. Available: https://www.cncf.io/blog/2020/02/26/cncf-tools-overview-fluentd-unified-logging-layer/. [Último acceso: Julio 2022].

[86] S. Nayak, «Medium,» 3 Octubre 2020. [En línea]. Available: https://betterprogramming.pub/microservice-observability-metrics-bd9be270bc62. [Último acceso: Julio 2022].

[87] P. Krishna, «E2E Cloud,» 4 Marzo 2021. [En línea]. Available: https://www.e2enetworks.com/blog/the-why-how-and-what-of-metrics-and-observability. [Último acceso: Julio 2022].

[88] C. Mair y M. Shepperd, «Human judgement and software metrics: Vision for the future,» *Proceedings - International Conference on Software Engineering,* 2011.

[89] E. Moraes, «Medium,» 4 Abril 2019. [En línea]. Available: https://medium.com/oracledevs/how-to-keep-your-microservices-available-by-monitoring-its-metrics-d88900298025. [Último acceso: Julio 2022].

[90] M. Herring, «InfluxData,» 19 Mayo 2022. [En línea]. Available: https://www.influxdata.com/time-series-database/. [Último acceso: Julio 2022].

[91] S. Nayak , «FreeCodeCamp,» 30 Diciembre 2020. [En línea]. Available: https://www.freecodecamp.org/news/microservice-observability-metrics/. [Último acceso: Julio 2022].

[92] M. Sitzenstatter y S. Kupstaitis-Dunkler, «Grafana,» 15 Febrero 2022. [En línea].

Available: https://grafana.com/blog/2022/02/15/what-are-cardinality-spikes-and-why-do-they-matter/. [Último acceso: Julio 2022].

[93] B. Sigelman, «Lighstep,» 13 Mayo 2020. [En línea]. Available: https://lightstep.com/apm#apm-vs-observability. [Último acceso: Julio 2022].

[94] AppDynamics, «AppDynamics,» [En línea]. Available: https://www.appdynamics.com/topics/what-is-APM. [Último acceso: Julio 2022].

[95] A. Magnusson, «Strongdm,» 11 Mayo 2022. [En línea]. Available: https://www.strongdm.com/blog/observability-vs-monitoring. [Último acceso: Julio 2022].

[96] M. McGuinness, «ApplicationPerformance,» 17 Enero 2017. [En línea]. Available: https://blog.applicationperformance.com/a-brief-history-of-application-performance-management-apm. [Último acceso: Julio 2022].

[97] M. Novakovic, «Instana,» 7 Agosto 2015. [En línea]. Available: https://www.instana.com/blog/application-performance-management-apm-where-is-it-going/. [Último acceso: Julio 2022].

[98] Gartner, «Gartner,» 17 Septiembre 2019. [En línea]. Available: https://www.gartner.com/en/information-technology/glossary/application-performance-monitoring-apm. [Último acceso: Julio 2022].

[99] A. Patel, «Instana,» 26 Febrero 2020. [En línea]. Available: https://www.instana.com/blog/modernizing-apm-for-devops-and-beyond/. [Último acceso: Julio 2022].

[100] B. Beyer, C. Jones, J. Petoff y N. R. Murphy, Site Reliability Engineering, O'Reilly Media, Inc., 2016.

[101] Blameless Blog, «Blameless,» 29 Julio 2021. [En línea]. Available: https://www.blameless.com/sre/4-sre-golden-signals-what-they-are-and-why-they-matter. [Último acceso: Julio 2022].

[102] Klipfolio, «Klipfolio,» [En línea]. Available: https://www.klipfolio.com/resources/articles/kpi-dashboard-operational-metrics-top-10-guidelines. [Último acceso: Julio 2022].

[103] GAInsights, «GAInsights,» [En línea]. Available: https://www.ga-insights.com/alerts. [Último acceso: Julio 2022].

[104] T. Feron, «Medium,» 23 Septiembre 2021. [En línea]. Available: https://levelup.gitconnected.com/software-metrics-best-practices-b91c37d87c73. [Último acceso: Julio 2022].

[105] L. Cavero-Baptista, «Leapwork,» [En línea]. Available: https://www.leapwork.com/blog/how-to-do-application-performance-monitoring-best-practices. [Último acceso: Julio 2022].

[106] B. Gregg, «Performance Checklists for SREs,» de *SREcon 2016*, Santa Clara, CA, 2016.

[107] E. Bruschini, «Instana,» 24 Octubre 2016. [En línea]. Available: https://www.instana.com/blog/monitoring-microservices-part-iii-investigate-

troubleshooting-machine-intelligence/. [Último acceso: Julio 2022].

[108] M. Novakovic, «Instana,» 30 Agosto 2016. [En línea]. Available: https://www.instana.com/blog/application-performance-management-3-0-time-new-approach-apm/. [Último acceso: Julio 2022].

[109] E. Andrews y A. Lê-Quôc , «TheNewStack,» 5 Mayo 2015. [En línea]. Available: https://thenewstack.io/collecting-metrics-using-statsd-a-standard-for-real-time-monitoring/. [Último acceso: Julio 2022].

[110] Statsd, «Github,» [En línea]. Available: https://github.com/statsd. [Último acceso: Julio 2022].

[111] O. Pomel, «DataDog,» 7 Agosto 2013. [En línea]. Available: https://www.datadoghq.com/blog/statsd/#what-problem-does-statsd-solve. [Último acceso: Julio 2022].

[112] Netdata Team, «Netdata,» 3 Febrero 2021. [En línea]. Available: https://www.netdata.cloud/blog/introduction-to-statsd/. [Último acceso: Julio 2022].

[113] J. Volz y B. Rabenstein, «SoundCloud,» 26 Enero 2015. [En línea]. Available: https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud. [Último acceso: Julio 2022].

[114] J. Volz, «Prometheus,» 26 Enero 2016. [En línea]. Available: https://prometheus.io/blog/2016/01/26/one-year-of-open-prometheus-development/. [Último acceso: Julio 2022].

[115] Prometheus, «Prometheus,» 20 Junio 2022. [En línea]. Available: https://prometheus.io/docs/concepts/. [Último acceso: Julio 2022].

[116] Prometheus, «Prometheus,» 20 Junio 2022. [En línea]. Available: https://prometheus.io/docs/prometheus/latest/querying/basics/. [Último acceso: Julio 2022].

[117] Prometheus, «Prometheus,» 20 Junio 2022. [En línea]. Available: https://prometheus.io/docs/prometheus/latest/querying/operators/. [Último acceso: Julio 2022].

[118] E. Tullstedt, «Grafana,» 4 Febrero 2020. [En línea]. Available: https://grafana.com/blog/2020/02/04/introduction-to-promql-the-prometheus-query-language/. [Último acceso: Julio 2022].

[119] CNCF, «CNCF,» 9 Mayo 2016. [En línea]. Available: https://www.cncf.io/projects/prometheus/. [Último acceso: Julio 2022].

[120] Prometheus, «Prometheus,» 20 Junio 2022. [En línea]. Available: https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config. [Último acceso: Julio 2022].

[121] Y. Grinshteyn, «OpenSource,» 6 Noviembre 2019. [En línea]. Available: https://opensource.com/article/19/11/introduction-monitoring-prometheus. [Último acceso: Julio 2022].

[122] B. Plotka, «Prometheus,» 16 Noviembre 2021. [En línea]. Available: https://prometheus.io/blog/2021/11/16/agent/. [Último acceso: Julio 2022].

[123] M. Pracucci, «Grafana,» 8 Abril 2022. [En línea]. Available: https://grafana.com/blog/2022/04/08/how-we-scaled-our-new-prometheus-tsdb-grafana-mimir-to-1-billion-active-series/. [Último acceso: Julio 2022].

[124] «Wikipedia,» 15 Junio 2022. [En línea]. Available: https://en.wikipedia.org/wiki/Log-structured_merge-tree. [Último acceso: Julio 2022].

[125] InfluxDb, «InfluxData,» 2 Julio 2022. [En línea]. Available: https://docs.influxdata.com/influxdb/v2.3/reference/internals/storage-engine/. [Último acceso: Julio 2022].

[126] Timescaledb, «Timescale,» 24 Mayo 2022. [En línea]. Available: https://docs.timescale.com/timescaledb/latest/overview/core-concepts/distributed-hypertables/#distributed-databases-and-nodes. [Último acceso: Julio 2022].

[127] J. Lockerman y A. Kulkarni , «Timescale,» 22 Abril 2020. [En línea]. Available: https://www.timescale.com/blog/time-series-compression-algorithms-explained/. [Último acceso: Julio 2022].

[128] P. Prateek, «SigNoz,» 2 Septiembre 2022. [En línea]. Available: https://signoz.io/blog/observability-net/. [Último acceso: Septiembre 2022].

[129] M. Korolov, «TheNewStack,» 16 Junio 2021. [En línea]. Available: https://thenewstack.io/growing-adoption-of-observability-powers-business-transformation/. [Último acceso: Julio 2022].

[130] A. Parker, D. Spoonhower, J. Mace, B. Sigelman y R. Isaacs, Distributed Tracing in Practice, O'Reilly Media, Inc., 2020.

[131] L. Chockalingam, «CNCF,» 6 Agosto 2021. [En línea]. Available: https://www.cncf.io/blog/2021/08/06/what-is-opentelemetry-and-why-is-it-the-future-of-instrumentation/. [Último acceso: Julio 2022].

[132] M. Vizard, «DevOps,» 17 Marzo 2022. [En línea]. Available: https://devops.com/log-management-challenges-grow-with-observability-adoption/. [Último acceso: Julio 2022].

[133] J. Wallace, «Coralogix,» 2 Noviembre 2021. [En línea]. Available: https://coralogix.com/blog/introducing-log-observability-microservices/. [Último acceso: Julio 2022].

[134] M. Conran, «NetworkInsight,» 8 Junio 2022. [En línea]. Available: https://network-insight.net/2022/06/08/observability-and-controllability-issues-with-metrics/. [Último acceso: Julio 2022].

[135] Mordor Intelligence, «MordorIntelligence,» 2021. [En línea]. Available: https://www.mordorintelligence.com/industry-reports/application-performance-management-apm-market. [Último acceso: Julio 2022].

[136] S. Spees, «Honeycomb,» 6 Octubre 2021. [En línea]. Available: https://www.honeycomb.io/blog/observability-101-terminology-and-concepts/. [Último acceso: Julio 2022].

[137] A. Vondrak, «Honeycomb,» 12 Agosto 2021. [En línea]. Available: https://www.honeycomb.io/blog/why-observability-requires-distributed-column-

store/. [Último acceso: Julio 2022].

[138] B. Brewer, «VentureBeat,» 17 Enero 2022. [En línea]. Available: https://venturebeat.com/2022/01/17/4-emerging-trends-point-to-changes-in-the-observability-landscape-in-2022/. [Último acceso: Julio 2022].

[139] H. Calato, «Finanzen,» 10 Junio 2022. [En línea]. Available: https://www.finanzen.at/nachrichten/aktien/honeycomb-named-a-leader-in-2022-gartner-magic-quadrant-for-application-performance-monitoring-and-observability-1031522728. [Último acceso: Julio 2022].

[140] C. Majors, L. Fong-Jones y G. Miranda, Observability Engineering, O'Reilly Media, Inc., 2022.

[141] S. Goldberg, «DZone,» 11 Agosto 2020. [En línea]. Available: https://dzone.com/articles/deep-dive-newsql-databases. [Último acceso: Agosto 2022].

[142] TiDB, «PingCAP,» 13 Junio 2022. [En línea]. Available: https://docs.pingcap.com/tidb/stable/overview. [Último acceso: Agosto 2022].

BIBLIOGRAFIA DE FIGURAS

- **Figura 1.** Autoría propia.
- **Figura 2,3.** Google Technical Report dapper-2010-1. Abril 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. https://research.google/pubs/pub36356/
- **Figura 4.** Zipkin. (n.d.). Architecture. https://zipkin.io/pages/architecture.html
- **Figura 5,6.** Jaeger. (n.d.). Architecture. https://www.jaegertracing.io/docs/1.35/architecture/
- **Figura 7.** Dynatrace. Octubre 2021. What is OpenTelemetry? https://www.dynatrace.com/news/blog/what-is-opentelemetry-2/
- **Figura 8.** Autoría propia.
- **Figura 9.** Grafana Labs. Febrero 2022. What are cardinality spikes and why do they matter? https://grafana.com/blog/2022/02/15/what-are-cardinality-spikes-and-why-do-they-matter/
- **Figura 10.** InfluxData. Diciembre 2017. The Effect of Cardinality on Data Ingest Part 1. https://www.influxdata.com/blog/the-effect-of-cardinality-on-data-ingest-part-1/