Prototipo de sistema de observabilidad de microservicios backend para el proyecto de Renovación de los Sistemas de Información UIS (R.S.I).

Amin Esteban Barbosa Vargas y Camilo Ciro Orozco

Trabajo de Grado para optar al título de Ingeniero de Sistemas

Director

Emilio Justiniano Carcamo Troconis

MSc. Ingeniería de sistemas e informática

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Ingeniería de Sistemas

Bucaramanga

2024

Dedicatoria

Este proyecto va dedicado con mucho amor a nuestros familiares, amigos, maestros y compañeros que estuvieron presentes en cada etapa de este proceso de formación.

Agradecimientos

Este proyecto ha pasado por múltiples facetas debido al impacto que alguna personas han aportado a lo largo de su desarrollo.

Inicialmente queremos agradecer a los profesores de la escuela de Ingeniería de Sistemas porque gracias a ellos hemos podido avanzar a través de estos 5 años adquiriendo nuevos conocie-mientos que nos permiten desenvolvernos en la vida profesional.

A nuestros colegas y amigos de estudio, quienes nos brindaron su apoyo en todo momento. Entre ellos una especial mención a Duvan Ferney Vargas Martínez y a Daniel Felipe Jaimes Blanco por su soporte en la orientación de este proyecto de grado.

A la comunidad del proyecto R.S.I. por darnos la oportunidad de empezar nuestra vida profesional en la institución que nos formó como ingenieros, así mismo como permitirnos realizar este proyecto.

Tabla de Contenido

1. Planteamiento y Justificación del Problema **21**

2. Objetivos **23**

2.1. Objetivo General 23

2.2. Objetivos Especificos 23

3. Marco de referencia **25**

3.1. Marco teórico 25

3.1.1. Fundamentos de arquitectura de software 25

3.1.1.1. Sistema de información 25

3.1.1.2. Arquitectura de Software 25

3.1.1.3. Arquitectura de microservicios 25

3.1.1.4. Monitoreo y Observabilidad 25

3.1.1.5. Métricas, Logs y Trazas 26

3.1.1.6. DevOps 26

3.2. Estado del arte 26

3.2.1. Herramientas o soluciones de monitoreo y observabilidad open source 27

3.2.1.1. Monasca 27

3.2.1.2. OpenTelemetry 30

3.2.1.3. Grafana Labs 32
3.2.1.4. ELK stack 34
3.2.1.5. Apache SkyWalking 35
3.2.1.6. Prometheus 36
3.2.2. Herramientas o soluciones de monitoreo y observabilidad closed source 38
3.2.2.1. Instana 38
3.2.2.2. Splunk 39
3.2.2.3. Azure Monitor 40
3.2.2.4. AWS CloudWatch 42

4. Metodología 45
4.1. Investigación 45
4.2. Implementación 46
4.3. Pruebas 46
4.4. Evaluación y análisis de resultados 46

5. Desarrollo 47
5.1. Investigación 47
5.1.1. Conceptualización de Observabilidad 47
5.1.1.1. Sistema observable 47
5.1.1.2. Importancia de un sistema de observabilidad 48
5.1.1.3. Monitoreo vs Observabilidad 48

5.1.1.4. 3 pilares de la observabilidad 50

5.1.2. Diseño de arquitectura 52

5.1.3. Establecer métricas de evaluación para seleccionar herramientas 54

5.2. Implementación 57

5.2.1. Escenario 1: Implementación de arquitectura de monitoreo en una única máquina 59

5.2.2. Escenario 2: Aumento de aplicaciones basadas en microservicios en una única máquina 62

5.2.3. Escenario 3: Separación del sistema de observabilidad y la aplicación basada en microservicios en diferentes máquinas 63

5.2.4. Escenario 4: Aplicando el sistema de observabilidad sobre una aplicación de R.S.I. 65

5.2.5. Escenario 5: Evaluando la carga del sistema de observabilidad al cargar múltiples aplicaciones de R.S.I. 66

5.2.6. Escenario 6: Escalando la arquitectura del sistema de observabilidad 68

5.3. Pruebas 69

5.3.1. Simulación de múltiples usuarios 69

5.3.2. Servicios no disponibles 74

5.3.3. Evaluando el balanceador de cargas 76

5.4. Evaluación y análisis de resultados 79

6. Conclusiones 80

7. Trabajo Futuro 82

Referencias Bibliográficas **83**

Apéndices **85**

Lista de Figuras

Figura 1. Gráfico de la arquitectura de Monasca. 28

Figura 2. Gráfico de la arquitectura de OpenTelemetry. 31

Figura 3. Gráfico descripción general de Loki. 33

Figura 4. Gráfico descripción general de Tempo. 34

Figura 5. Gráfico Elastic Observability. 35

Figura 6. Gráfico de la arquitectura de Prometheus. 38

Figura 7. Gráfico de la vista general de Splunk Observability Cloud. 40

Figura 8. Gráfico de la arquitectura de Azure Monitor. 41

Figura 9. Gráfico descripción general de AWS CloudWatch 43

Figura 10. Gráfico de la arquitectura de AWS CloudWatch. 44

Figura 11. Trayectoria metodología. 45

Figura 12. Gráfico de la arquitectura de Monitor 52

Figura 13. Gráfico de la arquitectura implementando un Broker 53

Figura 14. Gráfico de la arquitectura implementando un balanceado de cargas 54

Figura 15. Gráfico de la arquitectura final con las herramientas seleccionadas 56

Figura 16. Gráfico de la conexión entre servicio de Springboot y Broker 59

Figura 17. Gráfico de la conexión entre las herramientasd de monitoreo y el visualizador 60

Figura 18. Gráfico de la arquitectura del escenario 1 de la implementación 61

Figura 19. Gráfico de la traza de una peticion tipo GET 61

Figura 20. Gráfico de la arquitectura del escenario 2 de la implementación 62

Figura 21. Gráfico de la vista general de la telemetria del escenario 2 en grafana 63

Figura 22. Gráfico de la arquitectura del escenario 3 de la implementación 63

Figura 23. Gráfico de las peticiones HTTP realizadas sobre la aplicación en el escenario 3 65

Figura 24. Gráfico de la arquitectura del escenario 4 de la implementación 65

Figura 25. Gráfico de la telemetría exportada por la aplicación de R.S.I. 66

Figura 26. Gráfico de la arquitectura del escenario 5 de la implementación 67

Figura 27. Gráfico del rendimiento de las herramientas en el escenario 5 de la implementación 67

Figura 28. Gráfico de la telemetría exportada por las múltiples aplicaciónes de R.S.I. 68

Figura 29. Gráfico de la arquitectura del escenario 6 de la implementación 69

Figura 30. Gráfico del plan de pruebas realizado en JMeter 70

Figura 31. Gráfico de los dashboards creados para visualizar los datos de las aplicaciones de SpringBoot 71

Figura 32. Gráfico de los logs de la aplicación de Contratación después de ejecutar el plan de pruebas de JMeter 71

Figura 33. Gráfico de los logs de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter 72

Figura 34. Gráfico de las trazas de la aplicación de Contratación después de ejecutar el plan de pruebas de JMeter 72

Figura 35. Gráfico de las trazas de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter 73

Figura 36. Gráfico de las métricas de la aplicación de Contratación después de ejecutar el

plan de pruebas de JMeter 73

Figura 37. Gráfico de las métricas de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter 74

Figura 38. Gráfico de los servicios activos en nuestro sistema de observabilidad 74

Figura 39. Gráfico error en Grafana al no tener datasources activas 75

Figura 40. Gráfico datos exportados cuando las herramientas de monitoreo se encontraban abajo 75

Figura 41. Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-grado-collector-1 se encuentra apagado 76

Figura 42. Gráfico de los datos exportados cuando proyecto-grado-collector-1 se encuentra apagado 76

Figura 43. Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-grado-collector2-1 se encuentra apagado 77

Figura 44. Gráfico de los datos exportados cuando proyecto-grado-collector2-1 se encuentra apagado 77

Figura 45. Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-

grado-collector-1 y proyecto-grado-collector2-1 se encuentran apagados 78

Figura 46. Gráfico de los datos exportados cuando proyecto-grado-collector-1 y proyecto-grado-collector2-1 se encuentran apagados 78

Lista de Tablas

Tabla 1. Tabla con los objetivos específicos y sus distintas secciones donde son solucionados. 24

Tabla 2. Tabla de diferencias entre monitoreo y observabilidad 49

Tabla 3. Tabla de las herramientas y su respectiva puntación según las métricas establecidas 55

Tabla 4. Tabla de las características de la máquina virtual utilizada 64

Lista de Apéndices

pág.

Apéndice 1. Repositorio que contiene el sistema de observabilidad 85

Apéndice 2. Tabla de las métricas para seleccionar herramientas de logs 85

Apéndice 3. Tabla de las métricas para seleccionar herramientas de métricas 86

Apéndice 4. Tabla de las métricas para seleccionar herramientas de trazas 87

Apéndice 5. Dockerfile en aplicación de SpringBoot de prueba (Java 17) 88

Apéndice 6. Enlace para descargar OpenTelemetry Java Agent 89

Apéndice 7. Dockerfile en aplicación de SpringBoot (Java 17) de prueba que ejecuta el OpenTelemetry Java Agent 89

Apéndice 8. Dockerfile en aplicación de SpringBoot (Java 11) de R.S.I. que ejecuta el OpenTelemetry Java Agent 90

Apéndice 9. docker-compose en aplicación de SpringBoot de prueba (Java 17) 91

Apéndice 10. sdk-config de aplicación de SpringBoot de prueba (Java 17) 92

Glosario

API: Son creaciones hechas por los desarrolladores para construir comunicaciones entre los sistemas.

APM: El monitoreo de rendimiento de aplicaciones (Application Performance Monitoring) es el proceso en el cual se analizan los datos de las aplicaciones para identificar y solucionar fallos.

Backend: Sesión donde se aloja la lógica del negocio, principalmente se caracteriza por comunicarse con las base de datos y actuar por detrás del sistema.

Contenedores: Se le puede considerar como un recipiente o un espacio de software donde se empaqueta el código junto con sus dependencias, esto, con el objetivo de que la aplicación se ejecute rápidamente en un ambiente informático sea cual sea.

Docker: Es un servicio que permite construir o desplegar imágenes de contenedores.

Escalabilidad: Se entiende como la capacidad de aplicación de un sistema para satisfacer necesidades empresariales.

Escalabilidad horizontal: Se define como el caso de agregar más instancias o tecnologías a una arquitectura sin impactar significativamente el uso de recursos de cómputo.

Escalabilidad vertical: Busca añadir más medios de cómputo, como memoria, a un mismo nodo del sistema.

Framework: En programación, se le conoce como un conjunto de herramientas y librerías que facilita la construcción de aplicaciones.

Frontend Parte de la aplicación que interactúa directamente con el cliente.

gRPC: Es un framework que ofrece una forma de enlazar servicios de forma eficaz proveyendo soportes para balanceo de cargas, rastreo, comprobación de estado, entre otras.

HTTP: Protocolo diseñado principalmente para la comunicación entre navegadores y servidores web.

JSON: Formato ligero de intercambio de datos.

Máquina virtual: Es un entorno virtual que funciona como sistema informático virtual con su propia CPU, memoria, interfaz de red donde sus recursos son apartados de la máquina del sistema de hardware.

Modelo Vista Controlador: Es un patrón de arquitectura donde separa la lógica de la aplicación y la interfaz o vista.

Open Source: En el mundo de la programación, un software es considerado open source si cual-

quier persona puede inspeccionar, alterar o enriquecer el código fuente.

PromQL: Lenguaje de consulta para los datos expuestos por Prometheus.

REST: Es conocida por ser un modelo arquitectónico de software que expone una serie de propiedades para el buen funcionamiento de una API.

RESTFUL: Es una interfaz que utilizan los sistemas de computación para hacer intercambios de información de forma segura por medio de Internet.

Saturación: Media de los porcentajes de uso del sistema, como por ejemplo, cuánta memoria o cpu está consumiendo el sistema.

Scraping: Técnica en la cual un programa informático extrae datos del resultado generado por otra aplicación.

Spring Boot: Es un framework Java basado en el Modelo Vista Controlador el cual ofrece un modelo de configuración basada para aplicaciones o cualquier tipo de despliegues.

Telemetría: La telemetría es la medición automática y la transmisión inalámbrica de datos provenientes de orígenes remotos.

Trazabilidad: Es una práctica de control que ayuda a obtener el producto en el dominio de la solución lo más exacto y fiable posible.

Resumen

Título: Prototipo de sistema de observabilidad de microservicios backend para el proyecto de Renovación de los Sistemas de Información UIS (R.S.I). *

Autores: Amin Esteban Barbosa Vargas, Camilo Ciro Orozco **

Palabras Clave: Monitoreo, observabilidad, métricas, logs, trazas, microservicios.

Descripción: La observabilidad de los sistemas es una necesidad actual para la detección de fallos y reportes sobre la salud del aplicativo; los equipos de desarrollo optimizan la resolución de las dificultades que se les presentan al tener acceso a los logs, métricas y trazas de un servicio garantizando de esta forma que se encuentren funcionales la mayor parte del tiempo. Este documento además de plantear la importancia de un sistema de observabilidad, se encarga de diseñar un modelo arquitectónico que pueda suplir las necesidades básicas de cualquier proyecto de desarrollo, además, de mostrar como escala al paso del crecimiento de la aplicación. Se inicia con un estudio premilinar sobre los conceptos que abarca el marco de la observabilidad, luego sobre las distintas herramientas existentes para la recolección y visualización de datos y finalmente diseñando una serie de arquitecturas donde se le realizan diferentes pruebas para probar su versatilidad, eficiencia y desempeño.

---

* Trabajo de grado

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Emilio Justiniano Carcamo Troconis

Abstract

Title: Microservices observability system prototype for the Information Systems Renewal project's backend at Industrial University of Santander (UIS). *

Authors: Amin Esteban Barbosa Vargas, Camilo Ciro Orozco **

Keywords: Monitoring, observability, metrics, logs, traces, microservices

Description: Observability is a current need for the detection of failures and reports on the health of the application; development teams may optimize the resolution of the difficulties they face by having access to logs, metrics and traces of a service, thus ensuring that they are up most of the time. This document explain the importance of an observability system and show an architectural model design that can meet the basic needs of any development project, as well as showing how it scales as the application grows. It starts with a preliminary study on the concepts covered by the observability framework, then on the different existing tools for data collection and visualization and finally designing a series of architectures where different tests are performed to prove its versatility, efficiency and performance.

* Bachelor Thesis

** Faculty of Physicomechanical Engineering. School of Systems and Computer Engineering. Director: Emilio Justiniano Carcamo Troconis

Introducción

La arquitectura de microservicios consiste en un conjunto de servicios autónomos y pequeños que permite manejar diferentes lógicas empresariales para solucionar requerimientos específicos donde cada uno de ellos realiza sus propias pruebas, gestiona sus dependencias y demás implementaciones relacionadas a su correcto funcionamiento, permitiendo así partir de pequeñas tareas a la integración de un conjunto global; de esta manera ya se tiene una aplicación escalable y flexible a cualquier tipo de cambio que se requiera. Sin embargo, si bien es cierto que con este tipo de modelos tenemos mayor control de los sistemas, se suele tener el problema de crear y usar microservicios de forma descontrolada, ocasionando que la complejidad del aplicativo también crezca y a su vez, hace más difícil mantener una depuración estable para cada uno de ellos.

Por consecuencia, se emplean métodos de monitoreo; supervisar los sistemas basados en microservicios es una tarea fundamental debido a que nos permite conocer cómo controlarlo, asegurarnos que sea fiable, se encuentre disponible y que funcione como se tiene imaginado. Por esa razón, el proyecto tiene como objetivo analizar e idear un prototipo para un sistema de observabilidad a nivel backend en aplicaciones basadas en microservicios que satisfaga los 3 pilares fundamentales de la observabilidad: métricas, logs y trazas, enfocado sobre un proyecto funcional: La Renovación de los Sistemas de Información (R.S.I.).

1. Planteamiento y Justificación del Problema

Las aplicaciones con arquitecturas basadas en microservicios son muy populares hoy en día y muchas empresas como Amazon, Netflix, Deutsche Telekom, LinkedIn, entre otras más, han adoptado este estilo arquitectural (de Vries, Sjouke and Blaauw, Frank and Andrikopoulos, Vasilios, 2023) en sus sistemas backend ya que ofrece la facilidad de crear múltiples lógicas de negocio independientes a través de API RESTful u otros medios.

Es importante también tener en consideración que la cantidad de servicios independientes en cada aplicación que utiliza esta arquitectura puede aumentar con el tiempo. Siempre y cuando los servicios estén disponibles y preparados para su uso, tener numerosos microservicios no debería presentar ningún problema. Sin embargo, ningún sistema informático está exento de fallos y, cuando se produce uno, los servicios involucrados permanecen inoperables hasta que se detecte y se resuelva el problema. Es por ese motivo que la prioridad en este tipo de casos es reducir el lapso de tiempo entre la ocurrencia del fallo y su resolución.

Para lo anterior, es necesario contar con alguna herramienta que permita observar y analizar la aplicación deseada. Acercándonos un poco a contextos reales, proyectos de desarrollo como lo es R.S.I ya cuentan con una herramienta llamada “Monitor”[1], que se encarga de plasmar en un gráfico los logs generados por las diferentes aplicaciones desarrolladas en Spring Boot. Dicha herramienta suple de información constatemente a los programadores del área, detallando sobre errores que pueden llegar a suceder en algún microservicio, entre otros tipos de datos; teniendo esto en cuenta, se tiene como objetivo implementar un prototipo que permita el monitoreo de los logs, trazas y métricas generadas por los servicios, que informe sobre la medición del estado interno del sistema para la prevención o identificación de fallos y además que pueda adaptase a cuaquier aplicativo. Este prototipo se utilizará en proyectos funcionales

[1] Ejemplo de dasboard proyectado por Grafana: https://grafana.com/grafana/dashboards/

de R.S.I. donde se recibe la telemetría que exportan algunos de sus proyectos e identificar sus estados.

En resumen, la implementación de conceptos de observabilidad mejora las tácticas de escalado de los equipos DevOps, utilizando los datos de telemetría para definir una estrategia y llevar a cabo una planificación de recursos más efectiva. Finalmente, surgirán preguntas sobre el desarrollo del sistema de observabilidad tales como: ¿Qué factores de riesgo específicos pueden afectar la eficacia del sistema de observabilidad en un entorno de microservicios y cuáles son las estrategias más efectivas para mitigarlos? ¿Cómo evoluciona y se adapta la arquitectura de observabilidad en un proyecto de microservicios a medida que este crece y se desarrolla?

2. Objetivos

2.1. Objetivo General

- Diseñar un prototipo para un sistema de observabilidad que permita la recolección y visualización de logs, métricas y trazas de aplicaciones backend basadas en microservicios. Utilizando como caso de estudio el proyecto de Renovación de Sistemas de Información R.S.I.

2.2. Objetivos Especificos

- Investigar sobre los distintos conceptos que se deben tener en cuenta a la hora de prototipar un sistema de observabilidad.
- Realizar un estudio preliminar sobre la infraestructura arquitectónica de monitoreo manejada actualmente por el proyecto de R.S.I. con el fin de establecer una serie de requerimientos a tener en cuenta sobre las herramientas deseadas.
- Hacer una revisión de herramientas para el monitoreo de microservicios y seleccionar a las que hacen parte de los requerimientos deseados.
- Implementar una prueba piloto del sistema de observabilidad sobre un proyecto backend de R.S.I.

Tabla 1

Tabla con los objetivos específicos y sus distintas secciones donde son solucionados.

Objetivo	Cumplimiento
Investigar sobre los distintos conceptos que se deben tener en cuenta a la hora de prototipar un sistema de observabilidad.	El cumplimiento de este objetivo se evidencia en el capítulo 3 y la sección 5.1
Realizar un estudio preliminar sobre la infraestructura actual del proyecto de R.S.I para establecer una serie de métricas a tener en cuenta sobre las herramientas deseadas.	El cumplimiento de este objetivo se evidencia en la sección 5.1.2
Hacer una revisión de herramientas para el monitoreo de microservicios y seleccionar las que mejor se adapten a las métricas propuestas.	El cumplimiento de este objetivo se evidencia en la sección 3.2 y en las tablas 2,3 y 4.
Implementar una prueba piloto del sistema de observabilidad sobre un proyecto backend de R.S.I.	El cumplimiento de este objetivo se evidencia en el capítulo 5

3. Marco de referencia

3.1. Marco teórico

A continuación se presentan los conceptos que se requieren conocer para poder cumplir los objetivos propuestos en el trabajo de investigación

3.1.1. Fundamentos de arquitectura de software

3.1.1.1. Sistema de información. Se le conoce como aquellos grupos de componentes altamente relacionados con el objetivo de recolectar, procesar, almacenar y/o distribuir información para dar soporte a los procesos de decisiones y de control de una organización (LAUDON, KENNETH C. and LAUDON, JANE P, 2016).

3.1.1.2. Arquitectura de Software. Brinda un plan que permite desarrollar y diseñar aplicaciones, indicando las prácticas, patrones y técnicas que se van a emplear para poder conseguir un resultado bien estructurado (Red Hat, 2023a).

3.1.1.3. Arquitectura de microservicios. Es un estilo arquitectónico que estructura una aplicación como una colección de servicios que son desplegables de forma independiente, de acoplamiento flexible, organizados en torno a las necesidades del negocio y creados por un equipo pequeño (Richardson, C, 2023).

3.1.1.4. Monitoreo y Observabilidad. Monitoreo y Observabilidad son dos términos que a menudo se utilizan como sinónimos. Sin embargo, hay ligeras diferencias: El monitoreo es un proceso para el seguimiento de la salud del sistema, usando un conjunto predefinido de métricas y logs para buscar una colección de errores específicos. Por otro lado, la observabilidad es la habilidad de medir el estado interno del sistema a través de las salidas o external outputs, conocidos como métricas, logs y trazas. Esto permite identificar problemas que el monitoreo no puede

solucionar. Podemos decir entonces que el monitoreo es un prerrequisito para la observabilidad (Usman, Muhammad and Ferlin, Simone and Brunstrom, Anna and Taheri, Javid, 2022).

3.1.1.5. Métricas, Logs y Trazas. Los logs son líneas de texto que el sistema genera cuando se ejecuta una parte específica del código, dejando así un registro de un evento que ocurre dentro de nuestra aplicación. Esto permite prever comportamientos impredecibles y emergentes mostrados por los microservicios (Usman, Muhammad and Ferlin, Simone and Brunstrom, Anna and Taheri, Javid, 2022).

Las métricas son las representaciones numéricas de los datos que el equipo de operaciones utiliza para determinar el comportamiento de un sistema, servicio o componentes de red a lo largo del tiempo. A diferencia de los logs, las métricas son valores medidos a través del rendimiento en el tiempo de ejecución del sistema (Usman, Muhammad and Ferlin, Simone and Brunstrom, Anna and Taheri, Javid, 2022).

Las trazas representan el recorrido que realiza una solicitud o acción a través de los componentes del sistema (Usman, Muhammad and Ferlin, Simone and Brunstrom, Anna and Taheri, Javid, 2022).

3.1.1.6. DevOps. El término proviene de juntar las palabras “development” (desarrollo) y .ºperations"(operaciones), sin embargo, su filosofía representa un significado mucho más amplio de la unión de dichos conceptos, después de todo, llega a incluir sistemas de seguridad, metodología de trabajo, análisis de datos, entre otros; en pocas palabras, es una forma para denominar una cultura que se encarga de automatizar y diseñar plataformas para tener un mejor impacto valorativo en la organización, además de ofrecer respuestas rápidas y constantes de los servicios de TI (Red Hat, 2023b).

3.2. Estado del arte

El monitoreo de aplicaciones no es una problemática nueva. Sin embargo, la evolución que han tenido los desarrollos de software implica que las soluciones de monitoreo no se apliquen de igual manera en todos los casos, además de volver algunas soluciones obsoletas en periodos muy cortos de tiempo. En esta sección revisaremos las

soluciones que ofrecen parcial o totalmente soluciones de observabilidad para sistemas basados en microservicios. Incluiremos soluciones tanto open-source como closed-source.

3.2.1. Herramientas o soluciones de monitoreo y observabilidad open source

Algunas de las soluciones o herramientas open source más relevantes encontradas son las siguientes:

3.2.1.1. Monasca. Monasca es un aplicativo de código abierto que se encarga de realizar tareas de monitoreo, además, cuenta con características tales como ser altamente escalable, eficiente, multiusuario y tiene la ventaja de tolerar fallos que se integra con OpenStack. Utiliza una API REST para procesar y consultar métricas a alta velocidad y cuenta con un motor de alarmas y un motor de notificaciones en streaming (Monasca, 2015).

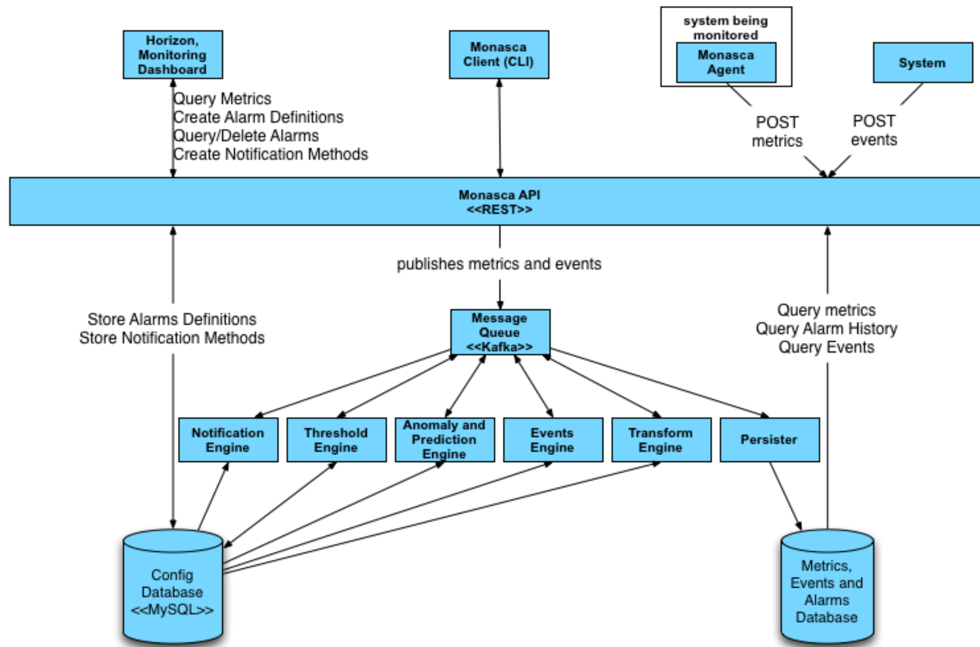
Monasca ofrece los siguientes funcionalidades:

- Una solución de monitorización como servicio (MONaaS) escalable que se ajusta según las métricas expuestas por los servicios. Es capaz de procesar miles de métricas por segundo y tenerlas por grandes cantidades de tiempo sin pérdida alguna.
- API de REST para almacenamiento y consulta de las métricas. Al ser HTTP el único protocolo disponible, se reduce el diseño lo cual permite una descripción detallada de los datos a través de las dimensiones.
- Multiusuario y autenticado.
- Las métricas definidas en conjunto de pares (clave, valor) denominados dimensiones.
- Umbrales y alarmas de las métricas en tiempo real.
- Alarmas compuestas descritas en sintaxis simple compuesta por subexpresiones de alarma y operadores lógicos.
- Agente de supervisión.
- Solución de monitoreo de código abierto basada en tecnologías de código abierto.

En la siguiente figura se aprecia la arquitectura de Monasca:

Figura 1.

Gráfico de la arquitectura de Monasca.



Nota: Componentes de la arquitectura de Monasca. Tomado de: Monasca, 2015

- Monitoring Agent (monasca-agent): Agente de monitorización basado en Python que soporta métricas del sistema, tales como la utilización de la cpu y la memoria actual utilizada, entre otros servicios integrados para herramientas como MySQL, RabbitMQ y muchos otros.
- Monitoring API (monasca-api): Una API RESTful rica en documentación para el monitoreo de servicios que se centra en los siguientes conceptos y áreas:
  - Métricas.
  - Estadísticas.
  - Definición de Alarmas.

   - Consulta de Alarmas.
   - Procedimientos de notificación.
   - Disponible en Java y Python.

- Persister (monasca-persister): Consume las métricas y los estados de las alarmas que brinda el MessageQ y las almacena en la base de datos de Métricas y Alarmas.
- Transform and Aggregation Engine (monasca-transform): Envía al Message Queue nuevas métricas al cambiar los nombres y valores de las antiguas métricas.
- Anomaly and Prediction Engine: Predice las anomalías a través de la evaluación de predicciones generada por métricas predichas.
- Threshold Engine (monasca-thresh): Calcula y asigna los umbrales sobre las métricas para ser publicadas por alarmas en el Message Queue cuando se exceden.
- Notification Engine (monasca-notification): Envía las notificaciones dadas por el Message Queue de los mensajes de transición de estado de las alarmas.
- Analytics Engine (monasca-analytics): Realiza la detección de anomalías y agrupación/correlación de alarmas.
- Message Queue: Recibe las métricas dadas por la API de monitoreo y mensajes de estado de alarmas desde el Threshold Engine. Actualmente se trabaja con terceros tales como un Message Queue basado en Kafka.
- Metrics and Alarms Database: Almacena las métricas y el historial de estado de las alarmas. Es un componente de terceros que soporta InfluxDB, Vertica y Cassandra.
- Config Database: Almacena la configuración y otro tipo de información relacionada del sistema. Componente de terceros que soporta MySQL
- Monitoring Client (python-monascaclient): Cliente de línea de comandos y biblioteca en Python que comunica y controla la API de monitoreo.

- Monitoring UI: Un tablero para visualizar la salud y estado general de una nube de OpenStack
- Ceilometer publisher: Plugin multipublicador para Ceilometer que convierte y publica muestras en la API de monitoreo.

3.2.1.2. OpenTelemetry. OpenTelemetry es un proyecto de la Cloud Native Computing Foundation (CNCF) resultado de la fusión de dos proyectos anteriores, OpenTracing y OpenCensus. Ambos proyectos se crearon para resolver el mismo problema: la falta de un estándar sobre cómo instrumentar código y enviar datos de telemetría a un backend de observabilidad. Como ninguno de los dos proyectos era totalmente capaz de resolver el problema de forma independiente, se fusionaron para formar OpenTelemetry y combinar sus puntos fuertes ofreciendo una única solución. Entonces podemos decir que es un framework y conjunto de herramientas de observabilidad diseñado para crear y gestionar datos de telemetría como trazas, métricas y logs. Fundamentalmente, es independiente del proveedor y de la herramienta, lo que significa que se puede utilizar con una amplia variedad de backends de observabilidad, incluyendo herramientas de código abierto como Jaeger y Prometheus, así como ofertas comerciales (OpenTelemetry, 2022).

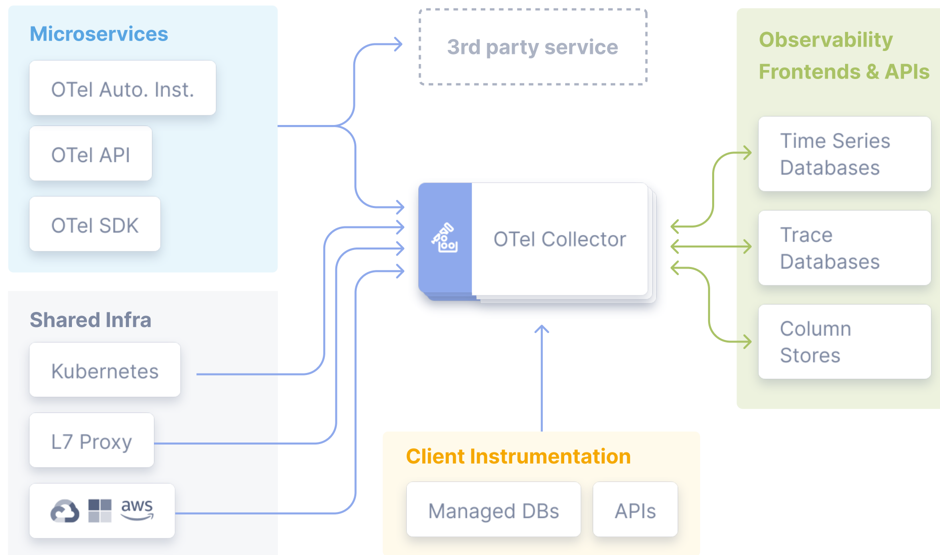
OpenTelemetry no es un backend de observabilidad como Jaeger, Prometheus, u otros proveedores comerciales. Se centra en la generación, recogida, gestión y exportación de telemetría. Un objetivo importante de OpenTelemetry es que se puede instrumentar fácilmente sus aplicaciones o sistemas, independientemente de su idioma, infraestructura o entorno de ejecución. El almacenamiento y la visualización de la telemetría se deja intencionalmente a otras herramientas.

OpenTelemetry satisface la necesidad de observabilidad al tiempo que sigue dos principios clave: usted es dueño de los datos que genera, no hay dependencia del proveedor y usted sólo tiene que aprender un único conjunto de API y convenciones. Además cuenta con los siguientes componentes principales:

- Una especificación para todos los componentes

Figura 2.

Gráfico de la arquitectura de OpenTelemetry.



Nota: Componentes de la arquitectura de OpenTelemetry. Tomado de: OpenTelemetry, 2024

- Convenciones semánticas que definen un esquema de nomenclatura estándar para los tipos de datos telemétricos más comunes.
- API que definen cómo generar datos telemétricos.
- Lenguajes SDK que implementan la especificación, las API y la exportación de datos telemétricos.
- Un ecosistema de bibliotecas que implementa la instrumentación para bibliotecas y marcos de trabajo comunes.
- Componentes de instrumentación automática que generan datos telemétricos sin necesidad de modificar el código.
- OpenTelemetry Collector, un proxy que recibe, procesa y exporta datos de telemetría.
- Varias otras herramientas, tales como el Operador OpenTelemetry para Kubernetes, OpenTelemetry Helm Charts, y los activos de la comunidad para FaaS

3.2.1.3. Grafana Labs. La empresa Grafana Labs ha creado su propio ecosistema de observabilidad con múltiples herramientas encargadas de cada aspecto.

Grafana es un software de código abierto que se encarga de representar y analizar datos métricos. Además, presenta distintas funcionalidades como consultar, alarmar y explorar sus métricas, registros y trazas, independientemente de dónde se encuentren guardados. Por otro lado, contiene instrumentos que le permiten transformar los datos de su base de datos de series temporales (TSDB) en gráficos y visualizaciones perspicaces (Grafana Labs, 2014).

Loki es un sistema encargado de recopilar eventos, es decir logs. Se distingue por ser escalable horizontalmente, altamente disponible, multi inquilino y basado o inspirado en Prometheus. A diferencia de este, Loki se enfoca principalmente en logs y no en métricas, además de recopilar los registros mediante push en vez de pull. En comparación con otras herramientas, Loki no indexa el contenido de los datos, en lugar de ello, solo los metadatos son indexados como una agrupación de etiquetas para cada flujo de registros.

Figura 3.

Gráfico descripción general de Loki.

Nota: Arquitectura básica con la implementación de Loki. Tomado de: Grafana Labs, s.f.-b

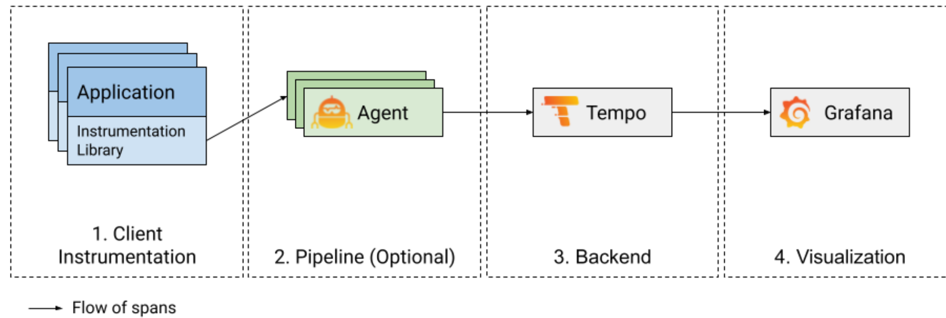
Un stack de logs basada en Loki consta de 3 componentes:

- Agente: Encargado de recolectar la información de los eventos, después, tiene la tarea de pasar dichos datos a flujos donde se le incluirá etiquetas y exporta los registros a Loki por medio de una API HTTP. De ejemplos como agentes se tiene Promtail o Grafana Agent.
- Loki: El componente principal, responsable de la ingestión y almacenamiento de registros y el procesamiento de consultas.
- Grafana: Para poder realizar búsquedas y tener representaciones gráficas de los registros.

Tempo es un servicio de código abierto encargado de la lógica del trazado, sencillo de implementar, con la capacidad de controlar grandes cantidades de información de forma eficiente y solo necesita de un almacenamiento de objetos para funcionar. Tiene la ventaja de complementarse muy bien con otros software como Grafana, Loki, Prometheus, Mimir y de implementarse con protocolos de trazado de código abierto como Jaeger, Zipkin u Open-Telemetry. Tempo maneja un lenguaje de consultas de trazas basado en LogQl y PromQl, llamado TraceQL; dicho lenguaje tiene la capacidad de realizar consultas que provee al usuario una forma sencilla y precisa de seleccionar los spans requeridos.

Figura 4.

Gráfico descripción general de Tempo.

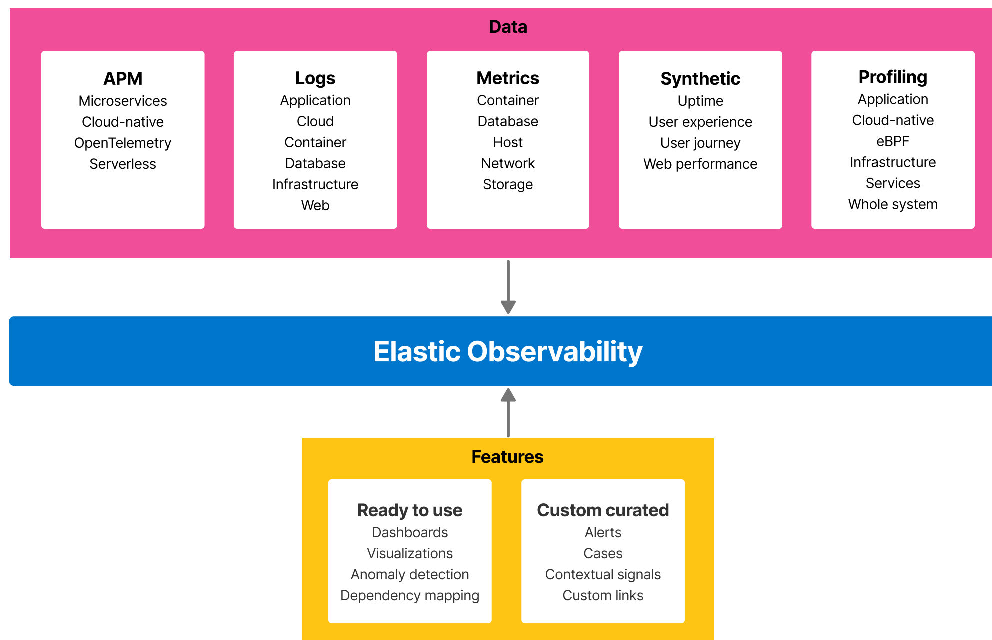


Nota: Arquitectura básica con la implementación de Tempo. Tomado de: Grafana Labs, s.f.-a

3.2.1.4. ELK stack. La empresa Elastic proporciona un stack para unificar sus logs, métricas de infraestructura, trazas de aplicaciones, datos de experiencia de usuario, sintéticos y perfilado universal. Ingesta sus datos directamente en Elasticsearch, donde puede procesar y mejorar aún más los datos antes de visualizarlos y agregar alertas en Kibana.

Figura 5.

Gráfico Elastic Observability.



Nota: Vista general de los componentes que contiene Elastic Observability. Tomado de: Elastic, 2013

3.2.1.5. Apache SkyWalking. Apache SkyWalking es una plataforma de observabilidad de código abierto diseñada para monitorear y diagnosticar sistemas distribuidos y aplicaciones en entornos de nube y microservicios (Apache SkyWalking, 2017). Apache SkyWalking ofrece las siguientes funcionalidades:

- Monitorización distribuida: SkyWalking es capaz de monitorear sistemas distribuidos, lo que significa que puede rastrear y recopilar datos de múltiples servicios y componentes que interactúan entre sí en entornos distribuidos, como aplicaciones basadas en microservicios.
- Trazabilidad de extremo a extremo: Proporciona trazabilidad de extremo a extremo para solicitudes y transacciones en sistemas distribuidos. Esto permite seguir el camino de una solicitud a través de diferentes servicios y componentes, lo que facilita la identificación y el diagnóstico de problemas en la aplicación.
- Recopilación de métricas y telemetría: SkyWalking recopila una variedad de métricas y telemetría, incluidos

datos de rendimiento, tiempo de respuesta, latencia, errores y más, para proporcionar una visión completa del comportamiento y el estado de la aplicación.

- Detección de dependencias: Identifica automáticamente las dependencias entre los servicios y componentes de una aplicación, lo que ayuda a comprender cómo interactúan entre sí y a visualizar la topología de la aplicación.
- Visualización y análisis: SkyWalking ofrece herramientas de visualización y análisis que permiten a los usuarios explorar y analizar los datos recopilados de manera intuitiva. Esto incluye paneles de control, gráficos, diagramas de flujo y más para comprender mejor el rendimiento y el comportamiento de la aplicación.
- Compatibilidad con múltiples lenguajes y frameworks: Es compatible con una variedad de lenguajes de programación y frameworks, lo que permite monitorear aplicaciones desarrolladas en diferentes tecnologías y stacks tecnológicos.

3.2.1.6. Prometheus. Prometheus es una herramienta de código abierto que se encarga de monitorear los sistemas y generar alertas en caso de presentarse alguna novedad, originalmente desarrollada en SoundCloud. Desde su creación en 2012, Prometheus se ha convertido en un instrumento bastante apetecido e implementado en los proyectos de distintas organizaciones, esto a su vez, lo ha favorecido con una gran comunidad que promueve su mantenimiento. Actualmente es un framework que se encuentra alojado a la Cloud native computing Foundation en 2016, siendo el segundo proyecto después de Kubernetes, esto garantiza que se mantendrá independiente de cualquier empresa (Prometheus, 2014).

Prometheus tiene como trabajo recopilar, guardar, medir y etiquetar datos de series temporales, en otras palabras, registra el momento en el tiempo en el cual fue recolectado el dato y le añade una etiqueta para identificarlo.

Las principales funcionalidades de Prometheus son:

- Modelo de datos multidimensional: Utiliza un modelo de datos multidimensional donde los datos de series temporales se identifican por el nombre de la métrica y pares de clave/valor.

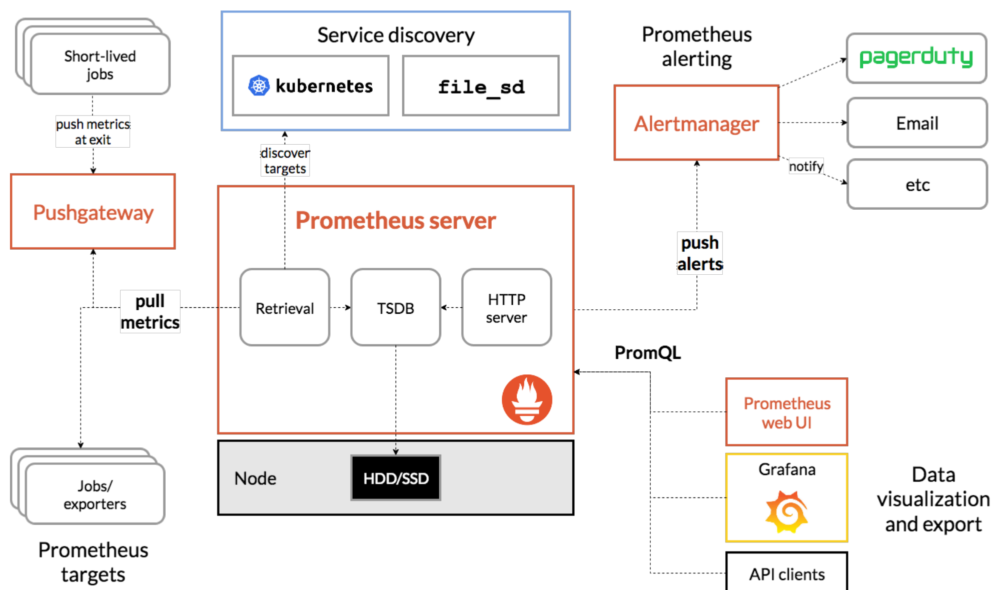
- PromQL: Proporciona un lenguaje de consulta flexible llamado PromQL para aprovechar esta dimensionalidad, permitiendo realizar consultas avanzadas sobre los datos.
- Independencia de almacenamiento distribuido: No depende de almacenamiento distribuido; los nodos del servidor son autónomos, lo que facilita su implementación y administración.
- Recolección de series temporales: La recolección de series temporales se realiza mediante un modelo de extracción (pull) a través de HTTP, lo que simplifica la configuración y operación del sistema.
- Soporte para envío de series temporales: Se admite el envío de series temporales a través de una pasarela (gateway) intermedia, lo que brinda flexibilidad en la arquitectura de monitoreo.
- Descubrimiento de objetivos: Los objetivos (targets) son descubiertos automáticamente a través de la autodetección de servicios o mediante configuración estática, lo que simplifica la gestión de infraestructuras dinámicas.
- Soporte para múltiples modos de visualización y creación de paneles: Prometheus ofrece varios modos de graficación y creación de paneles de control, lo que facilita la visualización y el análisis de los datos de monitoreo.

El ecosistema de Prometheus consiste en múltiples componentes, muchos de los cuales son opcionales:

- El servidor principal de Prometheus, que realiza el raspado (scraping) y almacena datos de series temporales.
- Bibliotecas de cliente para instrumentar el código de la aplicación, permitiendo la recopilación de métricas y su envío al servidor de Prometheus.
- Una pasarela de envío (push gateway) para admitir trabajos de corta duración que no pueden ser raspados directamente por el servidor principal.
- Exportadores especializados para servicios como HAProxy, StatsD, Graphite, etc., que permiten recopilar métricas específicas de estos servicios y enviarlas a Prometheus.
- Un administrador de alertas (alertmanager) para gestionar las alertas generadas por Prometheus y notificar a los equipos relevantes cuando ocurren eventos importantes.

Figura 6.

Gráfico de la arquitectura de Prometheus.



Nota: Tomado de Prometheus, 2014

- Varias herramientas de soporte, que pueden incluir herramientas de visualización, herramientas de análisis de datos, y otras utilidades que complementan la funcionalidad principal de Prometheus.

3.2.2. Herramientas o soluciones de monitoreo y observabilidad closed source

Algunas de las soluciones o herramientas closed source más relevantes encontradas son las siguientes:

3.2.2.1. Instana. IBM aporta a las soluciones de Observabilidad con su herramienta IBM Instana. Instana otorga constantemente datos de alta fidelidad en intervalos de 1 segundo y trazabilidad de las dependencias lógicas y físicas entre las aplicaciones móviles y web y la infraestructura (IBM, 2021). Instana brinda:

- Mejora en menor tiempo: La automatización de procesos que permiten comprender el contexto de la aplicación acelera la innovación del sistema.

- Optimización de operaciones para automatizar: Al utilizar soluciones basadas en IA se puede reducir el impacto de usuarios al predecir su comportamiento.
- Experiencia de usuario: Al brindar mejores experiencias de usuario estimula a los clientes a mantener el uso constante de su aplicación.
- Observabilidad democratizada: Instana no discrimina ningún tipo de usuarios avanzados. Ofrece a todos los profesionales en el ámbito de desarrollo de software los datos requeridos para el contexto necesario.

3.2.2.2. Splunk. La empresa Splunk ofrece su solución de observabilidad: Splunk Observability Cloud. Esta herramienta otorga una visión completa de las aplicaciones monitoreadas que permiten solucionar problemas en infraestructura, aplicaciones e interfaces de usuario (Splunk, 2004). Lo anterior mencionado se realiza en tiempo real lo que permite:

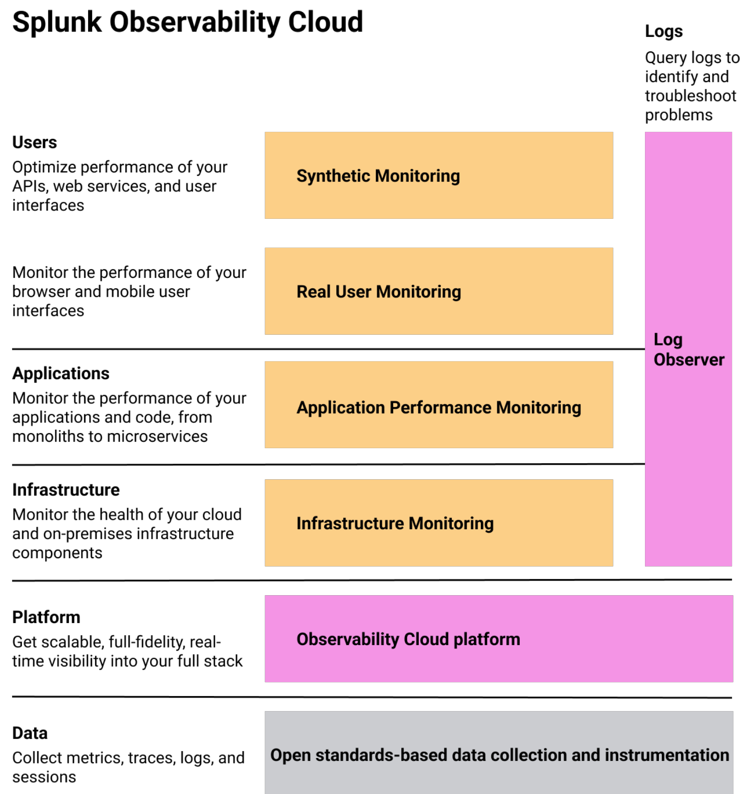
- Servicios confiables.
- Satisfacción por parte de los clientes.
- Innovar

Splunk nos ofrece más de 100 integraciones compatibles con Splunk Observability Cloud para obtener datos tanto en local y como la nube, en las aplicaciones o servicios, y en interfaces de usuario en Observability Cloud. Cuando se envía datos desde cada capa del entorno full-stack Observability Cloud, se transforman métricas, trazas y registros sin procesar en información accionable en forma de paneles de control, visualizaciones, alertas y más.

Las características que ofrece Splunk Observability Cloud le permiten a los clientes responder ágilmente a los obstáculos presentados e identificar el origen de los mismos. También brinda una gestión fundamentada en datos para aumentar la eficiencia y eficacia del sistema en el futuro.

Figura 7.

Gráfico de la vista general de Splunk Observability Cloud.



Nota: Tomado de Splunk, 2004

3.2.2.3. Azure Monitor. Azure Monitor es una solución de supervisión completa ofrecida por la empresa Microsoft que se encarga de recopilar, analizar y responder a los datos de supervisión de sus entornos en la nube y locales. Se puede usar Azure Monitor para maximizar la disponibilidad y el rendimiento de las aplicaciones y los servicios. También le ayuda a comprender cómo funcionan las aplicaciones y le permite responder manualmente y mediante programación a los eventos del sistema (Microsoft Azure, 2017).

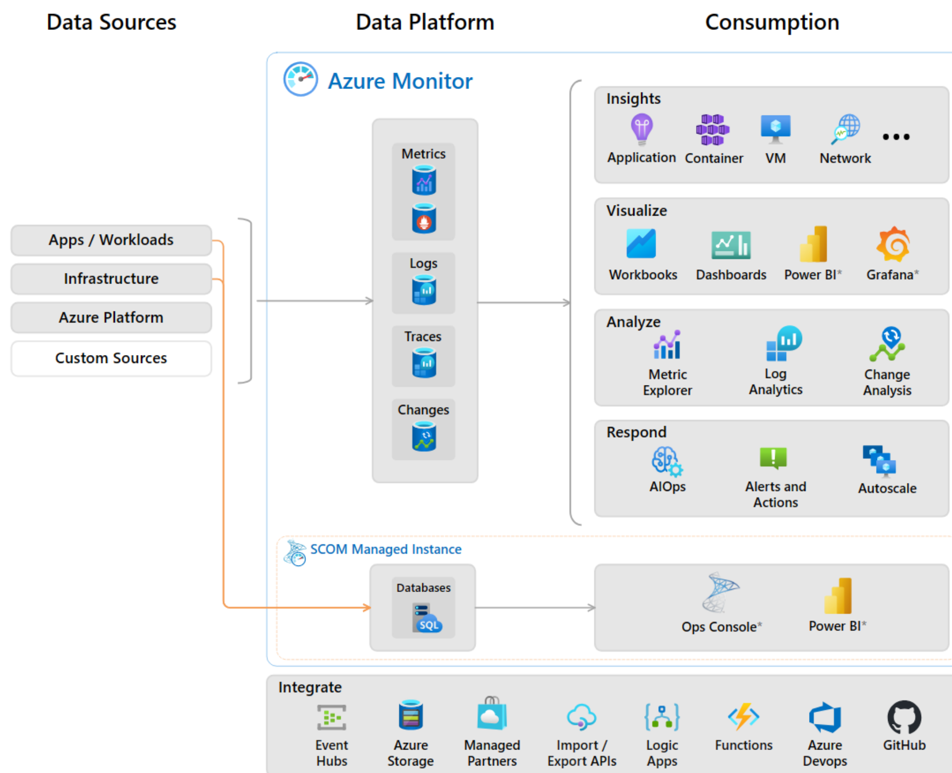
Azure Monitor recopila y agrega los datos de cada capa y componente del sistema en varias suscripciones e inquilinos que no son de Azure. Los almacena en una plataforma de datos común para su consumo por parte de un conjunto de herramientas que pueden correlacionar los datos, analizarlos, visualizarlos o responder a ellos. También

puede integrar otras herramientas ya sean o no elaboradas por Microsoft.

Azure Monitor puede supervisar estos tipos de recursos en Azure, en otras nubes o en el entorno local aplicaciones, máquinas virtuales, sistemas operativos invitados, contenedores con métricas de Prometheus, bases de datos, eventos de seguridad en combinación con Azure Sentinel, eventos de red y estado en combinación con Network Watcher y orígenes personalizados que usan las API para obtener datos.

Figura 8.

Gráfico de la arquitectura de Azure Monitor.



Nota: Tomado de Microsoft Azure, 2017

En el diagrama se muestran los componentes del sistema de Azure Monitor:

- Orígenes de datos son los tipos de recursos que se supervisan.

- Los datos se recopilan y enrutan a la plataforma de datos.
- La plataforma de datos almacena los datos de supervisión recopilados. La plataforma principal de datos de Azure Monitor tiene almacenes para métricas, registros, seguimientos y cambios. System Center Operations Manager MI usa su propia base de datos hospedada en SQL Managed Instance.
- En la sección consumo se muestran los componentes que usan datos de la plataforma de datos.
    - Los métodos de consumo principal de Azure Monitor incluye herramientas para proporcionar información, visualizar y analizar datos. Las herramientas de visualización se basan en las herramientas de análisis y la información se basa en las herramientas de visualización y análisis.
    - Hay mecanismos adicionales que le ayudarán a responder a los datos de supervisión entrantes.

3.2.2.4. AWS CloudWatch. Es un sistema que ayuda a observar y monitorear las aplicaciones que se encuentren tanto en local como subidas en algún servicio de nube; entrega información sobre la salud del proyecto, se puede configurar para generar respuestas o alarmas ante eventualidades, se encarga de que el consumo de recursos sea eficiente, entre otras tareas (Amazon Web Services, 2018).

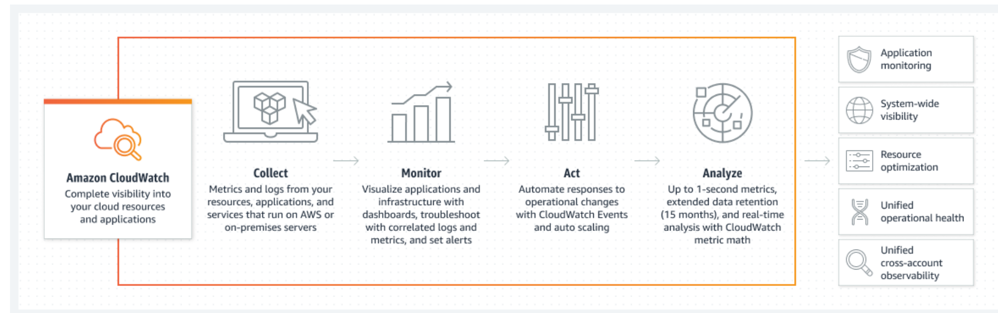
Ofrece los siguientes beneficios:

- Visualización y análisis de datos con observabilidad de principio a fin.
- Automatización de acciones y alarmas para activarse en umbrales determinados.
- Integración fácil con servicios de AWS.
- Gracias a la información proveniente de los registros y las métricas de los paneles de CloudWatch permite la solución de problemas operativos.

Amazon CloudWatch tiene la capacidad de recopilar las métricas, los registros, los datos de eventos en tiempo real, además, de poder plasmarlos en paneles para simplificar la infraestructura y el mantenimiento de aplicaciones.

Figura 9.

Gráfico descripción general de AWS CloudWatch

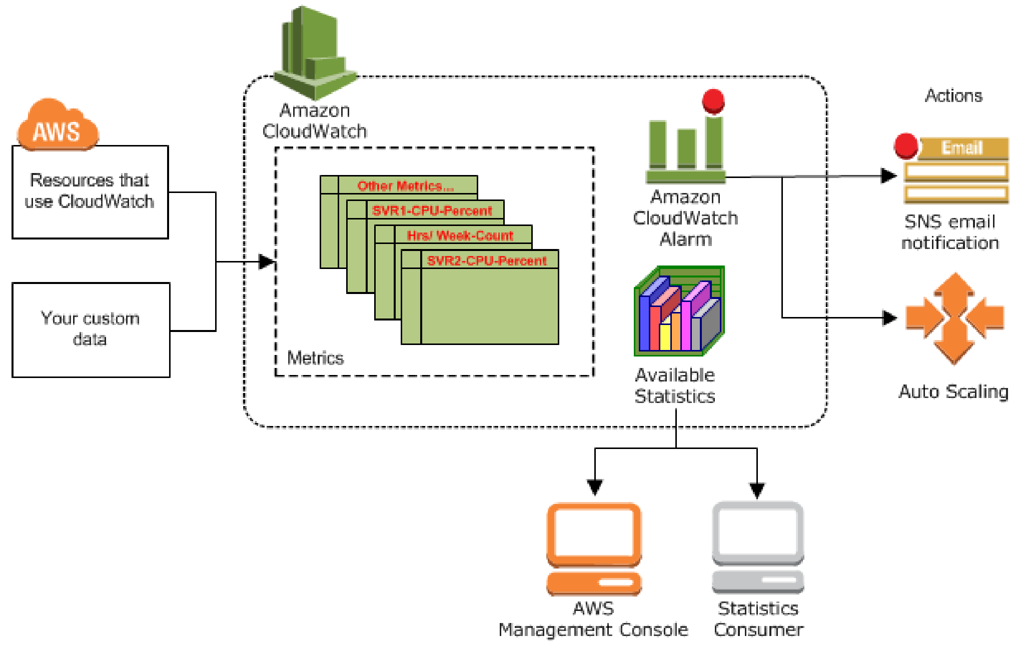


Nota: Tomado de Amazon Web Services, 2018

Amazon CloudWatch es básicamente una bodega de métricas. Un servicio de AWS, como lo es Amazon EC2, coloca métricas en el repositorio, y se recuperan estadísticas basadas en esas métricas. Se pueden colocar métricas personalizadas en el repositorio para recuperar estadísticas sobre estas métricas.

Figura 10.

Gráfico de la arquitectura de AWS CloudWatch.



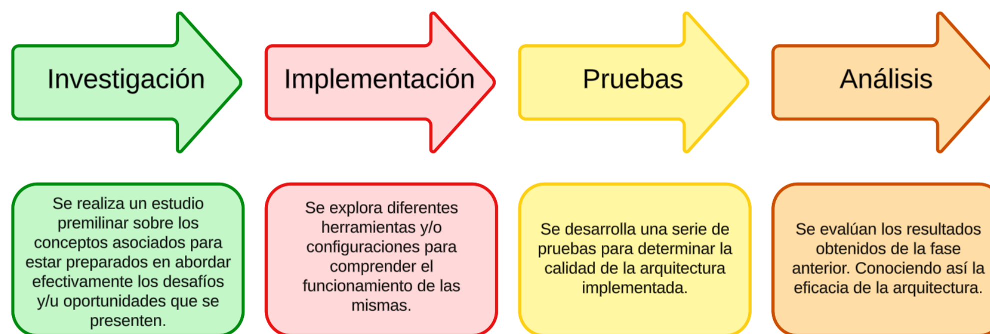
Nota: Tomado de Amazon Web Services, s.f.-a

4. Metodología

Un aspecto fundamental para la realización de este proyecto es buscar cómo abordar la temática de tal manera que se obtenga un resultado satisfactorio. Para esto, se establecen una serie de actividades que permiten avanzar en diferentes etapas del proyecto: investigación, implementación y análisis y conclusiones.

Figura 11.

Trayectoria metodología.



4.1. Investigación

Para esta etapa del proyecto, se busca recolectar datos que sean de todas las fuentes necesarias de utilidad, según los requerimientos planteados, para luego ser procesados y dar paso a la fase de implementación. Esta fase tiene gran impacto ya que la información obtenida se va a tomar como referencia para la realización del prototipo. Para realizar la investigación se tienen las siguientes actividades:

- Conceptualización de Observabilidad.
- Diseño de la arquitectura.
- Selección de herramientas.

4.2. Implementación

En la siguiente fase, se diseña un prototipo inicial del modelo de observabilidad utilizando distintas herramientas que tengan la capacidad de integrarse de manera efectiva, con el objetivo de suplir cualquier exigencia de los aplicativos que emplee microservicios. Las siguientes actividades hacen parte de la implementación:

- Implementación de múltiples escenarios de arquitectura basados en la arquitectura base.
- Implementación del prototipo del sistema de observabilidad con las herramientas seleccionadas.

4.3. Pruebas

En esta etapa se realizan las pruebas sobre el prototipo del sistema de observabilidad. Se tienen las siguientes pruebas:

- Simulación de múltiples escenarios
- Servicios no disponibles
- Evaluando el balanceador de cargas

4.4. Evaluación y análisis de resultados

En esta última etapa se recopilan todas las conclusiones obtenidas en las demás fases de la metodología, respondiendo las preguntas planteadas al inicio del desarrollo de este proyecto, así como preguntas que hayan surgido a través del mismo.

5. Desarrollo

Esta sección se enfoca en explorar detalladamente los procedimientos que sigue cada una de las etapas de la metodología que fue previamente introducida.

5.1. Investigación

Se busca recolectar datos de todas las fuentes necesarias que sean de utilidad para ser procesados y mantener una base sólida sobre los conceptos en cuestión permitiendo comprender cada aspecto que se realizará en la fase de implementación. Esta etapa se divide en subprocesos que permiten organizar y aclarar las ideas para la resolución del objetivo general de la misma.

5.1.1. Conceptualización de Observabilidad

En la conceptualización Observabilidad partimos desde la definición del término. Este punto de partida abre puertas a nuevos conceptos esenciales en el marco de la observabilidad a medida que avanzamos en la consulta de sus términos relacionados. Muchos de los términos se explican en las secciones de Glosario y Marco de referencia, por lo que en este documento nos enfocaremos en explicar el uso de los términos más relevantes a través de aplicaciones en problemas reales que guíen al cumpliento del objetivo general del proyecto de investigación.

5.1.1.1. Sistema observable. La observabilidad es un enfoque para comprender y visualizar de manera efectiva el comportamiento y el rendimiento de un sistema. Un sistema observable contiene las siguientes características principales:

- Posible inferir su estado interno basados en sus salidas (outputs).
- Provee información relevante sobre su funcionamiento interno que permite a los operadores y desarrolladores

diagnosticar problemas.

- La información dada por el sistema incluye datos como logs, métricas, trazas.

5.1.1.2. Importancia de un sistema de observabilidad. La observabilidad es un concepto importante en el desarrollo de software moderno, especialmente en el contexto de la arquitectura de microservicios. Los sistemas complejos se componen de muchos servicios diferentes que deben interactuar sin problemas. Sin embargo se debe tener presente que:

- Ningún sistema complejo está nunca completamente sano.
- Los sistemas distribuidos son patológicamente impredecibles.
- Es imposible predecir los innumerables estados de fallo parcial en los que pueden acabar partes del sistema.
- Hay que tener en cuenta los fallos en todas las fases, desde el diseño del sistema hasta la implementación, las pruebas, el despliegue y el mantenimiento.
- La facilidad de depuración es la piedra angular del mantenimiento y la evolución de los sistemas robustos.

El objetivo de la observabilidad es permitir una identificación y resolución más rápidas de los problemas, lo que en última instancia conduce a sistemas más fiables y eficientes.

5.1.1.3. Monitoreo vs Observabilidad. Tanto el Monitoreo como la Observabilidad se usan para mantener y administrar correctamente el estado y el rendimiento de las arquitecturas de microservicios distribuidos y su infraestructura. Para determinar cuándo es beneficioso utilizar uno sobre el otro en un sistema o aplicación se realizó un cuadro comparativo con algunas de sus características esenciales que permita un mejor entendimiento de las diferencias entre estos dos conceptos.

El monitoreo crea métricas, eventos, logs y rastreos descriptivos que miden lo que es esencial de una manera fácilmente identificable y recuperable. Los logs se almacenan junto con las mediciones actuales para crear una imagen

Tabla 2

Tabla de diferencias entre monitoreo y observabilidad

	Monitoreo	Observabilidad
¿Qué es?	Proceso de recopilar datos y generar informes sobre diferentes métricas que definen el estado del sistema.	Analiza detenidamente las interacciones de los componentes del sistema distribuido y los datos recopilados mediante el monitoreo para encontrar la causa raíz de los problemas.
Objetivo principal	Descubrir anomalías o comportamientos inusuales en el estado y el rendimiento del sistema.	Investigar más a fondo cualquier anomalía para encontrar la causa raíz de los problemas.
Sistemas involucrados	Por lo general, se ocupa de sistemas independientes.	Por lo general, se ocupa de sistemas múltiples y dispares.
Enfoque	Enfoque reactivo	Enfoque reactivo
¿Qué explica?	El *cuándo* y el *qué*.	El *por qué* y el *cómo*.

Nota: Diseño inspirado en Amazon Web Services, s.f.-b

amplia del sistema. La observabilidad luego puede utilizar el resultado del monitoreo para investigar los incidentes con mayor profundidad. Dicho esto, la observabilidad trae beneficios que el monitoreo por sí solo no logra satisfacer en su totalidad, como por ejemplo detectar la causa raíz de los problemas del sistema y evitar incidentes antes de que ocurran los problemas.

Sin embargo, no quiere decir que en el sistema el utilizar uno implica desechar al otro o, en otras palabras, que sean mutuamente excluyentes. La captura y el monitoreo de datos correctos, junto con la observabilidad, permiten rastrear errores a través de sistemas complejos

5.1.1.4. 3 pilares de la observabilidad. En un sistema que es muy demandado por muchas personas o usuarios, la resolución de problemas debe realizarse en el menor tiempo posible para evitar insatisfacción por parte de los clientes que utilizan el programa.

Cuando los logs, las métricas y las trazas trabajan juntos, revelan información profunda sobre el comportamiento del sistema, es por ello, que se le conocen como los tres pilares de la observabilidad. Aprovechar dichas entidades, se puede tener una compresión más exhaustiva sobre los problemas de rendimiento, identificar cuellos de botella y los microservicios que están causando retrasos que afectan a la experiencia general del cliente.

Los logs contienen detalles minuciosos sobre la etapa de procesamiento de las solicitudes de una aplicación. Capturan información detallada sobre eventos o transacciones individuales, ofreciendo un registro secuencial de lo sucedido. Las excepciones en los logs pueden proporcionar indicadores de problemas en una aplicación. La supervisión de errores y excepciones en los logs es una parte integral de una solución de observabilidad debido a que poseen las siguientes ventajas:

- Son fáciles de generar. Normalmente un timestamp y un payload.
- No necesitan integración explícita por parte de los desarrolladores además de una sentencia print.
- Texto sin formato y legible por humanos.
- La información detallada de los registros de aplicaciones o componentes individuales permite la reproducción retrospectiva de incidentes de soporte.

Las métricas son la representación numérica de un dato medido sobre un intervalo de tiempo. Normalmente, se accede a este tipo de datos operativos en tiempo real a través de una API mediante una estrategia pull o polling, o como un evento o telemetría generada, como un push o una notificación. La mayoría de las tareas de gestión de fallos están motivadas por métricas porque se basan en eventos.

Las métricas proporcionan datos agregados como los tiempos de respuesta o las tasas de error, lo que ofrece una visión de alto nivel del rendimiento del sistema. La recopilación de métricas suele ser intuitiva porque los indicadores de salud del sistema, como la CPU y la memoria son muy evidentes. Por lo tanto, determinar qué métricas recopilar continuamente y cómo analizarlas requiere mucho cuidado. Las métricas también contienen ventajas que vale la pena destacar:

- Son altamente cuantitativas e intuitivas para asociarlas con umbrales de alerta.
- Son ligeras de almacenar y recuperar.
- Permiten seguir tendencias a lo largo del tiempo y comprenderlas.
- Dan un entendimiento de cómo están cambiando los sistemas o servicios.

El concepto de Tracing (rastreo) es relativamente nuevo. Se trata de registros de flujos de trabajo creados para seguir un elemento de trabajo, como una transacción, a través de los pasos que la lógica de la aplicación le indica que siga. Una traza es una técnica indirecta para evaluar la complejidad de una aplicación, ya que la dirección del trabajo suele ser el resultado de la lógica de los componentes individuales o de herramientas de dirección como mallas de servicio o buses.

Para producir una traza, se agrupan los datos de múltiples componentes. Se muestra el flujo de trabajo de extremo a extremo de una única solicitud a través de un sistema distribuido permitiendo desglosar la latencia y asignarla a diferentes niveles o componentes, ayudando a identificar los cuellos de botella. Las ventajas que las trazas nos ofrecen son las siguientes:

- Son perfectas para identificar el componente o paso en el que se está produciendo el problema.
- Pueden depurar problemas en un sistema distribuido proporcionando un registro detallado del flujo de peticiones y respuestas.

- Ofrecen detalles específicos del contexto sobre el comportamiento del sistema, como la solicitud concreta que se está gestionando o el usuario que la ha realizado.

5.1.2. Diseño de arquitectura

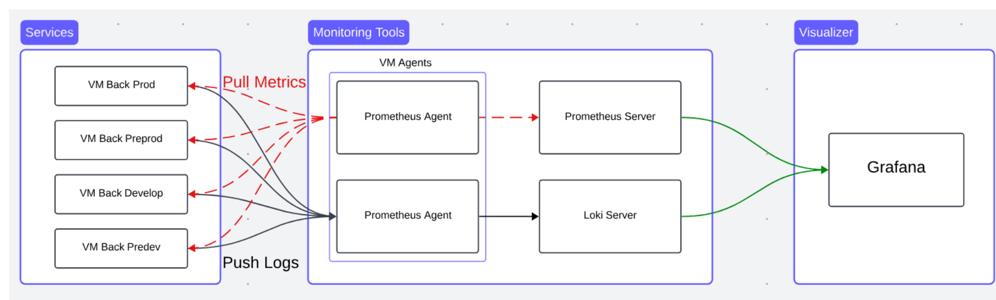
El siguiente paso en la investigación es diseñar la arquitectura base para un modelo de observabilidad.

Inicialmente se centra en conocer la arquitectura de la aplicación objetivo, específicamente en el componente backend. El proyecto de R.S.I. cuenta con cuatro (4) máquinas virtuales representando cada ambiente de desarrollo Predev, Develop, Preprod y Produccion, donde almacena sus aplicaciones de Springboot.

Luego, surge la necesidad de estudiar el funcionamiento de un sistema de monitoreo funcional que se utiliza en proyectos de desarrollo. Para esto, el proyecto R.S.I cuenta con su propio sistema de monitoreo llamado "Monitor". Junto con los encargados de Monitor se realiza un análisis del comportamiento del servicio y de la arquitectura planteada para la detección de problemas de las aplicaciones backend. Gracias a esto se logra tener una claridad sobre la manera que usan las herramientas implementadas en este sistema.

Figura 12.

Gráfico de la arquitectura de Monitor



Nota: En el gráfico se encuentran las herramientas de monitoreo Prometheus y Loki y el visualizador Grafana. Estas herramientas se tendrán en cuenta para la selección de herramientas en la siguiente fase.

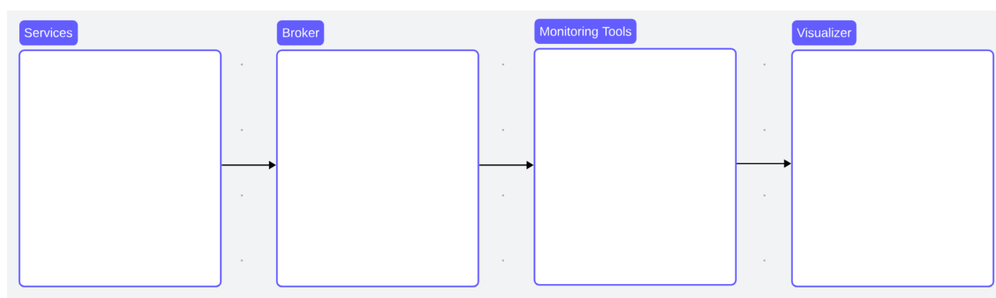
Se evidencia que el sistema cuenta con varios elementos claves: Los servicios, las herramientas de monitoreo

que exponen los datos de las aplicaciones y el visualizador que se encarga de recibir dicha información y representarla por medio de dashboards y gráficos con el fin de que sea más legible.

A la hora de investigar sobre arquitecturas de sistema de observabilidad, se evidencia el mismo patrón en todas ellas. El diseño inicial de nuestro prototipo es influenciado por este patrón, sin embargo, se adiciona un nuevo componente siguiendo el patrón arquitectural "Broker"[2], aportando así flexibilidad y adaptabilidad al sistema.

Figura 13.

Gráfico de la arquitectura implementando un Broker



La adición de un Broker permite estructurar sistemas de software distribuidos con componentes desacoplados que interactúan mediante llamadas a procedimientos remotos. Es decir, este componente se encarga de coordinar las comunicaciones tales como el reenvío de peticiones, la transmisión de los resultados y las excepciones.

Para el prototipo planteado, el Broker ofrece independencia entre las conexiones establecidas de los servicios monitoreados y las herramientas de monitoreo. Permite redistribuir las tareas mejorando la interacción de los componentes del sistema. También, la cantidad de microservicios podría aumentar fácilmente con el tiempo, lo cual facilita enlazar cada servicio con las diferentes herramientas de monitoreo utilizadas al servir como puente. Si es necesario cambiar o agregar una herramienta de monitoreo, solo hace falta que el broker lo incluya en su configuración.

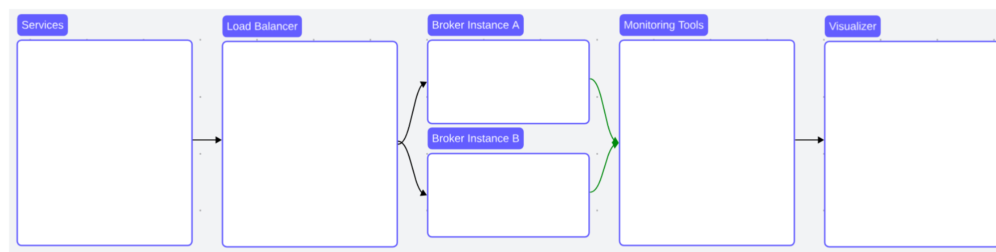
---

[2] Broker: Patrón de arquitectura que se utiliza como intermediario.

Adicionalmente, si se desea implementar un sistema de observabilidad sobre aplicaciones con alta demanda, es necesario contar con una implementación que permita aliviar las cargas de las peticiones realizadas al broker.

Figura 14.

Gráfico de la arquitectura implementando un balanceado de cargas



Para esta útlima arquitectura se decide utilizar un Load Balancer o Balanceador de Cargas por los beneficios que aporta al sistema. El Balanceador de Cargas recibe los datos de telemetría de las múltiples aplicaciones o servicios y los distribuye entre los diferentes objetivos (en este caso las instancias del Broker). Dirige las peticiones sólo a los objetivos que no se encuentren sobrecargados, garantizando que en momentos de mayor tráfico, aumente la fiabilidad de las aplicaciones. Por ejemplo, en dado caso que la instancia A del Broker no se encuentre disponible, el balanceador de cargas enviará los datos a la instancia B del Broker permitiendo que el sistema funcione con normalidad y que la instacnia A del Broker pueda recuperarse.

5.1.3. Establecer métricas de evaluación para seleccionar herramientas

Obtenida la arquitectura final a implementar, se avanza a el último paso de la fase de investigación: La selección de herramientas a utilizar. Para realizar esta escogencia, se diseñan unos requerimientos que deben cumplir las posibles herramientas como primera instancia:

- Cuente con una gran flexibilidad: Permite su uso en diferentes tipos de arquitecturas.
- No estén integradas en un sistema completo de observabilidad: La solución propuesta que se aborda en este documento no dependerá de un ecosistema ya establecido.

- Tolerante a fallos y alta disponibilidad: Cuenta con algunas funcionalidades que permitan el funcionamiento de la herramienta cuando ocurren problemas.
- Altamente escalable: La agregación de componentes al sistema no debe afectar la arquitectura, por el contrario, debe permitir el cambio sin problemas y el consumo de recurso computacional no debe verse incrementado drásticamente.

Pasado este filtro, las herramientas candidatas son comparadas según su funcionalidad, usabilidad, eficiencia, portabilidad y perspectiva de uso, a través de una serie de métricas establecidas para poder determinar las que mejores se acoplen a los requerimientos.

Tabla 3

Tabla de las herramientas y su respectiva puntación según las métricas establecidas

TRAZAS		
HERRAMIENTA	**PESO**	**RANGO**
Otel	184	1
Tempo	152	2
Jaeger	151	3
Zipkin	127	4
MÉTRICAS		
HERRAMIENTA	**PESO**	**RANGO**
Otel	184	1
Prometheus	172	2
Elastic	160	3
Graphite	130	4
LOGS		
HERRAMIENTA	**PESO**	**RANGO**
Otel	184	1
Loki	153	2
Logstash	128	5
Fluentd	152	3
Graylog	138	4

Nota: La comparativa de las herramientas se puede apreciar con mayor detalle en la sección de Apéndices.

Las herramientas escogidas finalmente son las siguientes:

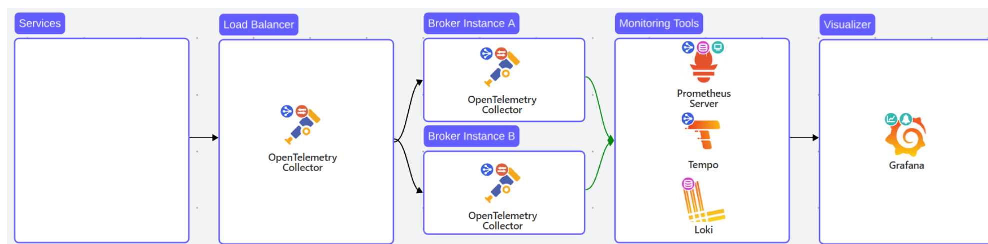
- Prometheus como backend de monitoreo de métricas
- Loki como backend de monitoreo de logs
- Tempo como backend de monitoreo de trazas
- Grafana como visualizador
- OpenTelemetry Collector como Broker

El Balanceador de Cargas no se tuvo en cuenta para esta selección de herramientas ya que lo ideal es implementarlo en sistemas distribuidos como Kubernetes. Para este proyecto, no se utiliza Kubernetes, sin embargo, OpenTelemetry Collector cuenta con una funcionalidad que le permite trabajar como balanceador de cargas por lo cual será implementado para esa tarea.

Tomando la arquitectura y las herramientas seleccionadas, se tiene el siguiente gráfico el cual será la guía para la fase de implementación:

Figura 15.

Gráfico de la arquitectura final con las herramientas seleccionadas



Nota: Las imágenes de las herramientas mostradas en esta figura y las siguientes son generadas en OpenAPM.io.

5.2. Implementación

Definidas las herramientas junto con la arquitectura base, se realiza la implementación del sistema de observabilidad.

Esta fase cuenta con una ruta experimental diseñada para apoyar la implementación y evaluar el escalado de la arquitectura del sistema de observabilidad para los microservicios a nivel backend. La motivación principal es evolucionar la arquitectura desde un punto inicial hasta alcanzar un nivel de configuración que refleje la complejidad y demás demandas actuales del entorno caracterizado por la adopción de una arquitectura de microservicios.

La elección de esta metodología de exploración experimental surge de la necesidad de comprender y analizar la tolerancia a cambios de la arquitectura del sistema de observabilidad y de qué manera logra adaptarse a medida que la infraestructura de los servicios en una organización experimenta un crecimiento, ya que es esencial que la arquitectura pueda escalar para seguir el ritmo del cambio.

Teniendo en cuenta lo anterior, se abordan varios aspectos fundamentales:

- Escalabilidad: La arquitectura debe contar con la capacidad de poder escalar horizontalmente, sobre todo, en entornos que presentan cambios constantemente.
- Optimización de recursos: En un entorno donde los recursos computacionales son valiosos, es esencial optimizar para garantizar un rendimiento óptimo del sistema. Observar cómo la arquitectura se comporta en diferentes configuraciones permitiendo identificar mejoras para optimizar el uso de recursos y la eficiencia del sistema.
- Tolerancia a fallos: En entornos donde los fallos pueden ocurrir en cualquier momento, es esencial que la arquitectura sea capaz de mantener el vínculo y la integridad de la observación.

Los siguientes escenarios son contemplados en la ruta experimental:

1. Arquitectura de monitoreo en una única máquina
2. Aumento de la carga de servicios en una única máquina
3. Separación de los servicios y la arquitectura en diferentes máquinas
4. Aplicando el sistema de observabilidad sobre un servicio de R.S.I
5. Evaluando la carga sobre el sistema sobre múltiples servicios de R.S.I
6. Escalando la arquitectura

Las herramientas seleccionadas de monitoreo son utilizadas así como también aplicaciones de SpringBoot (Java 11 y Java 17), VirtualBox, Git y Docker. Este último se elige como alternativa de contenerización debido a su reciente implementación en los proyectos de la R.S.I., además de facilitar los traslados a diferentes entornos similares de contenedores como Kubernetes.

Para el monitoreo de los microservicios se desea trabajar como monitoreo de caja negra, para no tener que realizar modificación en el código y se pueda implementar en cualquier aplicación deseada.

Por parte de la aplicación de Springboot, se agrega un agente de Java, OpenTelemetry Java Agent, que se activa en el momento de ejecución de nuestra aplicación. Este agente es el encargado de escuchar y exportar la telemetría ofrecida por el sistema de Java.

Por otro lado, en el sistema de observabilidad se contemplan varias instancias de OpenTelemetry Collector. Al recibir los datos por el agente de Java, el balanceador de cargas (una instancia de OpenTelemetry Collector) las reparte entre los diferentes broker existentes (también instancias de OpenTelemtry Collector). OpenTelemetry Collector contiene una funcionalidad que permite almacenar la información recibida hasta encontrar algún servicio al cual exportarlo, siendo útil para los momentos donde los backends de monitoreo no se encuentren disponibles. Adicionalmente, si se requiere, también cuenta con un Batch Processor que ayuda a comprimir mejor los datos y a reducir el

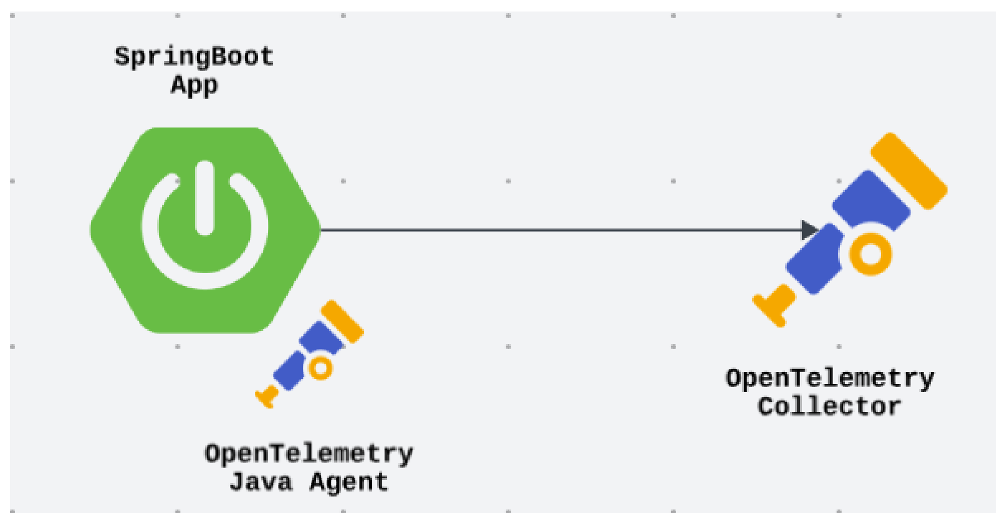
número de conexiones salientes necesarias para transmitirlos. Entonces, las herramientas dadas por OpenTelemetry van a cumplir un papel fundamental en este sistema de observabilidad.

5.2.1. Escenario 1: Implementación de arquitectura de monitoreo en una única máquina

El inicio de la fase de implementacion es un acercamiento de las herramientas seleccionadas. Para esto se establece una conexión entre la apliación de Springboot y el Broker.

Figura 16.

Gráfico de la conexión entre servicio de Springboot y Broker



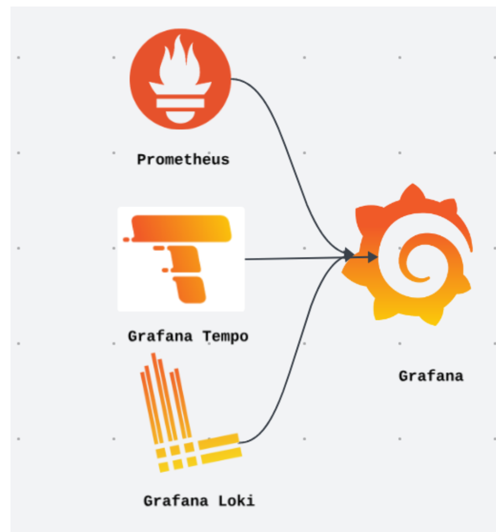
Nota: Los archivos de configuración se pueden encontrar en la sección de anexos.

A través de la configuración realizada, la telemetría enviada al Broker se visualiza a nivel de logs permitiendo comprobar que, efectivamente, estén conectadas.

Pasando a las herramientas de monitoreo, se establece la conexión entre ellas y el visualizador.

Figura 17.

Gráfico de la conexión entre las herramientasd de monitoreo y el visualizador



Nota: Los archivos de configuración se pueden encontrar en la sección de anexos.

La detección de las fuentes de datos (Prometheus, Tempo y Loki) en Grafana representa que se ha establecido correctamente la conexión. Con estos dos pasos introductorios, finalmente se puede avanzar con la ruta experimental con la arquitectura base.

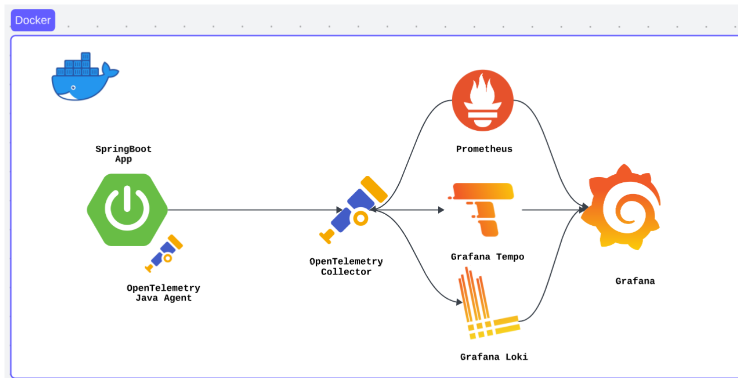
La funcionalidad se caracteriza principalmente en hacer que el proyecto de Spring boot cuente con un agente de java que permite recopilar información al ejecutarse el programa y enviarla al OpenTelemetry Collector, quien se encargará de repartir a cada herramienta según le corresponde la información obtenida para finalmente poder plasmar los datos o resultados hallados en Grafana.

La contenerización de la aplicación de Springboot junto con el agente de Java facilita la ejecución en simultáneo de estos dos (2) servicios, ya que es indispensable que ambos se encuentren activos para exportar los datos.

A este contenedor se le adicionan las herramientas de monitoreo a través de la funcionalidad de Docker: docker compose. En cada servicio se agregan las dependencias necesarias y variables de entorno de las imágenes.

Figura 18.

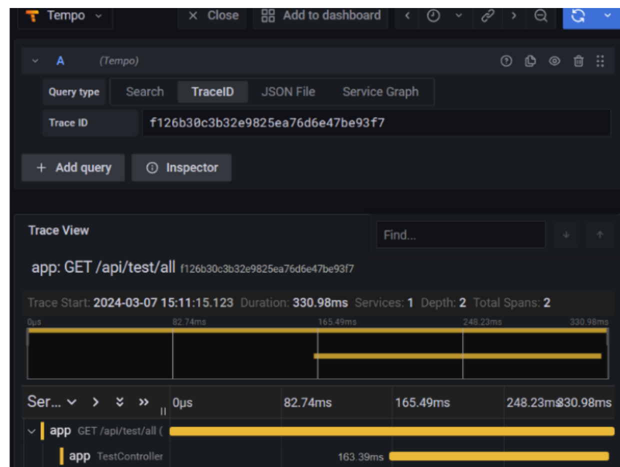
Gráfico de la arquitectura del escenario 1 de la implementación



El resultado en este escenario es poder relacionar la aplicación con las herramientas escogidas, de esta forma, se aprende a cómo trasladar la información entre los involucrados y así tener la capacidad de realizar mejoras de la arquitectura.

Figura 19.

Gráfico de la traza de una peticion tipo GET



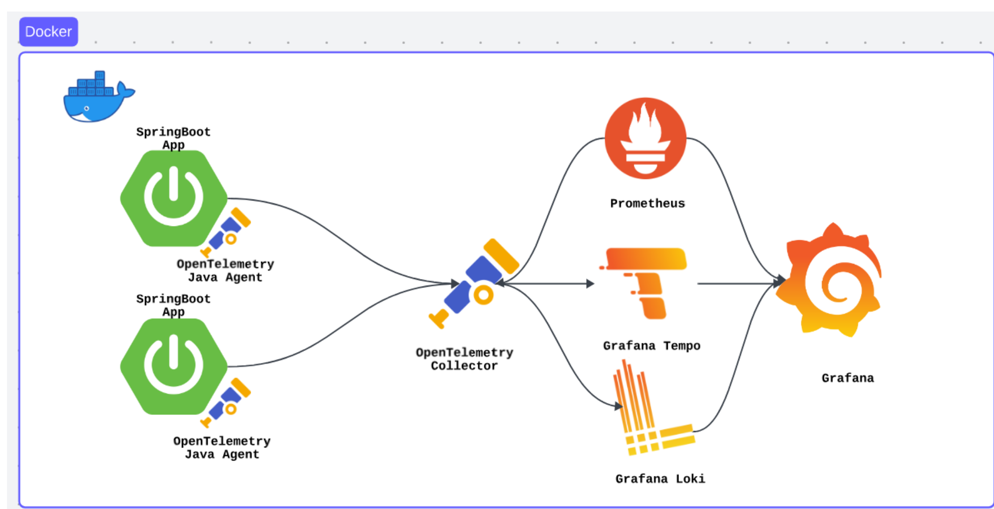
Este es un caso que solo se cumple al enlazar una única aplicación. En el siguiente escenario se prueba adicionando otra aplicación de SpringBoot.

5.2.2. Escenario 2: Aumento de aplicaciones basadas en microservicios en una única máquina

En este escenario, se introduce una sobrecarga al implementar un segundo servicio en la misma infraestructura. Esto permite evaluar cómo la arquitectura se comporta con múltiples servicios y si es capaz de mantener un rendimiento óptimo a pesar del crecimiento en la demanda de recursos.

Figura 20.

Gráfico de la arquitectura del escenario 2 de la implementación



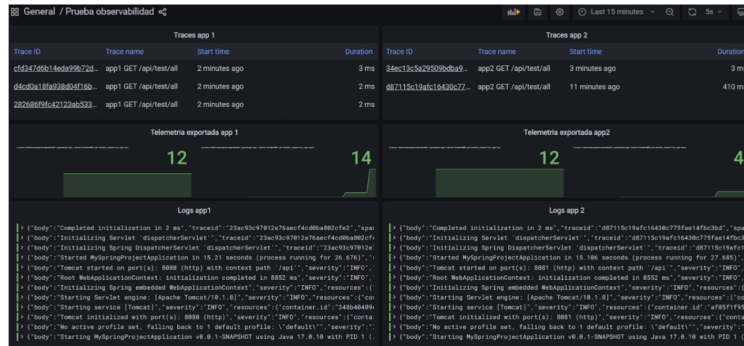
Para agregar la nueva aplicación al sistema de observabilidad, es necesario que la aplicación que se desea adicionar se encuentre en un contenedor o Dockerfile. Este contenedor se anexa al docker compose establecido en el escenario anterior (importante exponerla en un puerto diferente a la aplicación que ya se encontraba en el sistema).

Al realizar peticiones sobre ambos servicios, la telemetría se empieza a exportar al Collector y este a su vez envía a los diferentes backend de monitoreo.

El servicio de monitoreo logra soportar la carga de ambas aplicaciones y distingue entre la telemetría enviada por la aplicación 1 y la telemetría enviada por la aplicación 2.

Figura 21.

Gráfico de la vista general de la telemetria del escenario 2 en grafana

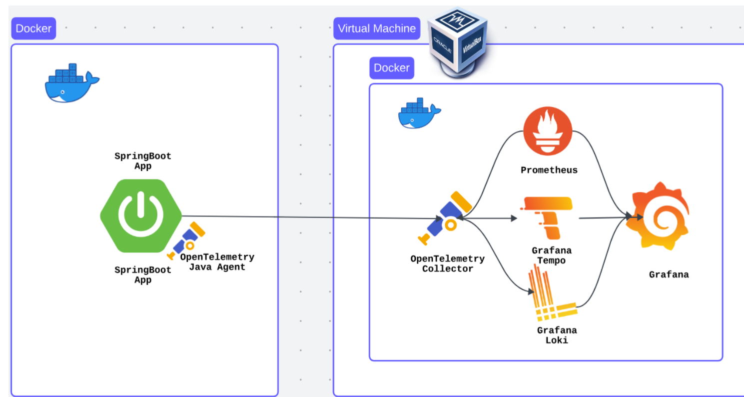


5.2.3. Escenario 3: Separación del sistema de observabilidad y la aplicación basada en microservicios en diferentes máquinas

En este escenario se realiza una adaptación de la infraestructura al dividirla en dos máquinas. Esto refleja que las aplicaciones monitoreadas se ejecutan en una máquina, mientras que la arquitectura de monitoreo se implementa en otra. Gracias a esta división se aliviana la sobrecarga en la infraestructura, además se evalúa cómo se comporta la arquitectura en un entorno distribuido.

Figura 22.

Gráfico de la arquitectura del escenario 3 de la implementación



Las máquinas virtuales permiten hacer esta división sin la necesidad de contar con múltiples máquinas físicas. A través de la herramienta VirtualBox se simula un servidor donde se almacena el sistema de observabilidad al cual la aplicación de SpringBoot realiza peticiones.

Tabla 4

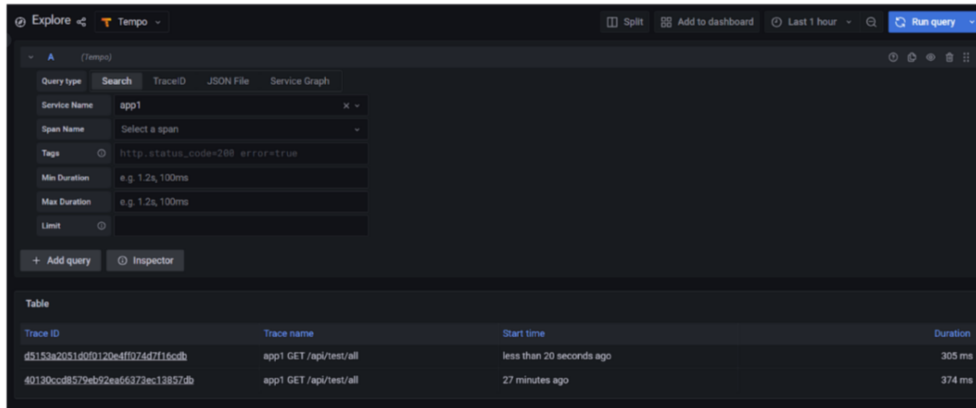
Tabla de las características de la máquina virtual utilizada

GENERAL	
Nombre	vm1
Sistema Operativo	Ubuntu server
SISTEMA	
Memoria base	2048 MB
Procesadores	2
Orden de arranque	Disquete, Óptica, Disco duro
Aceleración	Paginación anidada, Paravirtualización KVM
PANTALLA	
Memoria de video	16 MB
Controlador gráfico	VMSVGA

Anteriormente se hizo mención del motivo de implementación de la herramienta Docker. Para este caso el hacer la división de la arquitectura es tan sencillo como trasladar el contenedor hacia la máquina virtual y ejecutarlo. A través de la dirección IP de la máquina virtual se puede establecer la conexión de las aplicación de SpringBoot. El envío de la telemetría hacia y el tratamiento de los datos por parte del sistema de observabilidad se realiza satisfactoriamente.

Figura 23.

Gráfico de las peticiones HTTP realizadas sobre la aplicación en el escenario 3

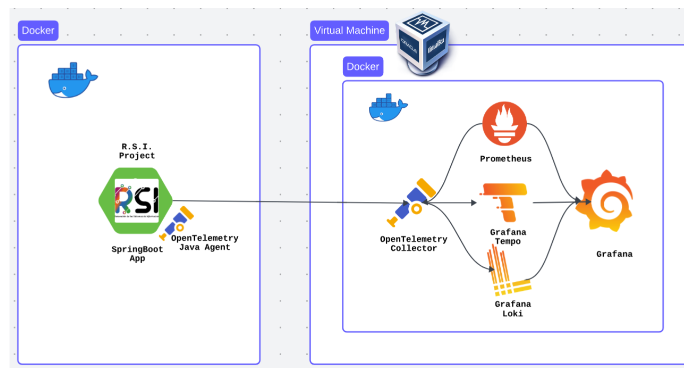


5.2.4. Escenario 4: Aplicando el sistema de observabilidad sobre una aplicación de R.S.I.

Para este punto, se tiene un sistema de observabilidad funcional para aplicaciones de SpringBoot. Ahora, se analiza si funciona correctamente con proyectos reales que muy posiblemente serán utilizados por usuarios en un ambiente de producción. En este escenario, se intercambian las aplicaciones de prueba que se contemplaban en las anteriores versiones por una aplicación del proyecto de la R.S.I., aportando complejidad a la infraestructura y sobrecarga en el sistema.

Figura 24.

Gráfico de la arquitectura del escenario 4 de la implementación



Repitiendo el procedimiento del primer escenario, se utiliza uno de los tantos servicios backend con los que cuenta la R.S.I. Arbitrariamente se toma el proyecto backend de Contratación (se puede utilizar cualquiera sin tener complicaciones con la configuración) y se ajusta el servicio para que se conecte a la máquina virtual permitiendo el envio de peticiones.

Con esto se determina que el sistema de observabilidad soporta proyectos reales que serán utilizados por usuarios en algún momento.

Figura 25.

Gráfico de la telemetría exportada por la aplicación de R.S.I.

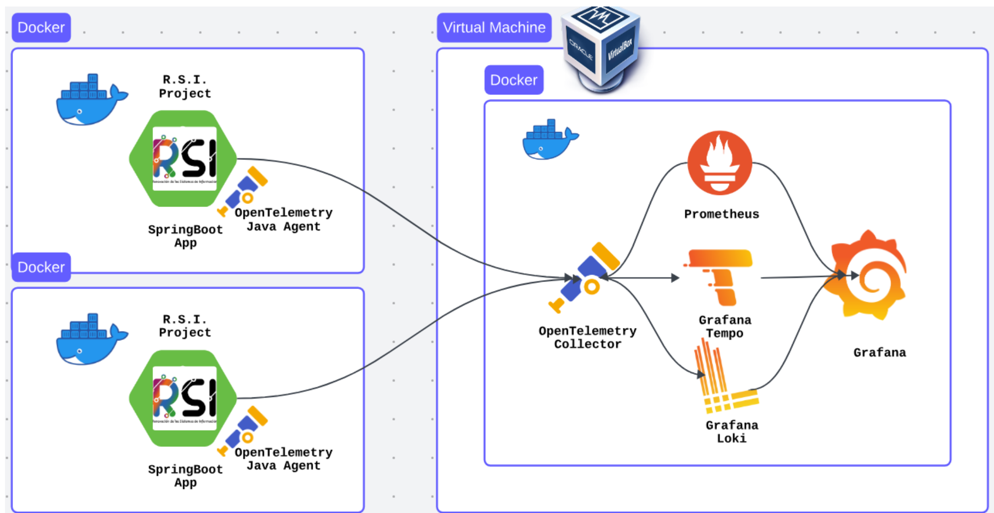
5.2.5. Escenario 5: Evaluando la carga del sistema de observabilidad al cargar múltiples aplicaciones de R.S.I.

La integración de un servicio backend del proyecto R.S.I. funciona correctamente. A continuacion se va agregar otro y se analiza el comportamiento del sistema.

Nuevamente se repiten los pasos del escenario anterior. Esta vez se utiliza el proyecto backend de Situaciones Administrativas. Este escenario explora la sobrecarga cuando múltiples servicios complejos hacen parte de la infraestructura y de que manera responde a esta carga de trabajo de servicios que pueden ser comunes en entornos de producción.

Figura 26.

Gráfico de la arquitectura del escenario 5 de la implementación



Este escenario es similar al escenario 2, por tanto el foco de esta implementación es un poco más técnico al probar el rendimiento cuando se realizan múltiples peticiones con la herramienta JMeter. Mientras JMeter se encuentra enviando las peticiones el consumo de recursos en la máquina virtual es el siguiente:

Figura 27.

Gráfico del rendimiento de las herramientas en el escenario 5 de la implementación

```
vm1 [Corriendo] - Oracle VM VirtualBox
Archivo  Máquina  Ver  Entrada  Dispositivos  Ayuda
CONTAINER ID   NAME                          CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
    BLOCK I/O         PIDS
93cbabcf08d0   grafana                       0.17%     61.76MiB / 1.918GiB   3.14%     8.99MB / 26.4M
B   174MB / 37.3MB    17
afb81ff45f69   prometheus                    5.92%     34.31MiB / 1.918GiB   1.75%     10.2MB / 2.65M
B   115MB / 23.7MB    9
146c5d04b326   proyecto-grado-collector-1    0.00%     51.8MiB / 1.918GiB    2.64%     8.32MB / 10.3M
B   301MB / 0B        8
c868014950cc   tempo                         26.62%    53.64MiB / 1.918GiB   2.73%     589kB / 244kB
    92.1MB / 50.7MB   9
ae9357659709   loki                          0.92%     51.45MiB / 1.918GiB   2.62%     265kB / 3.77MB
    117MB / 7.23MB    10
_
```

Después de enviar múltiples peticiones a través de JMeter, el uso de la CPU del sistema por el Collector, las peticiones HTTP que ha recibido y la telemetría exportada por cada aplicación son las siguientes:

Figura 28.

Gráfico de la telemetría exportada por las múltiples aplicaciónes de R.S.I.

Al obtener estos resultados se cerciora que el sistema de observabilidad logra mantenerse estable cuando múltiples servicios complejos envíen sus datos para ser tratados y visualizados.

5.2.6. Escenario 6: Escalando la arquitectura del sistema de observabilidad

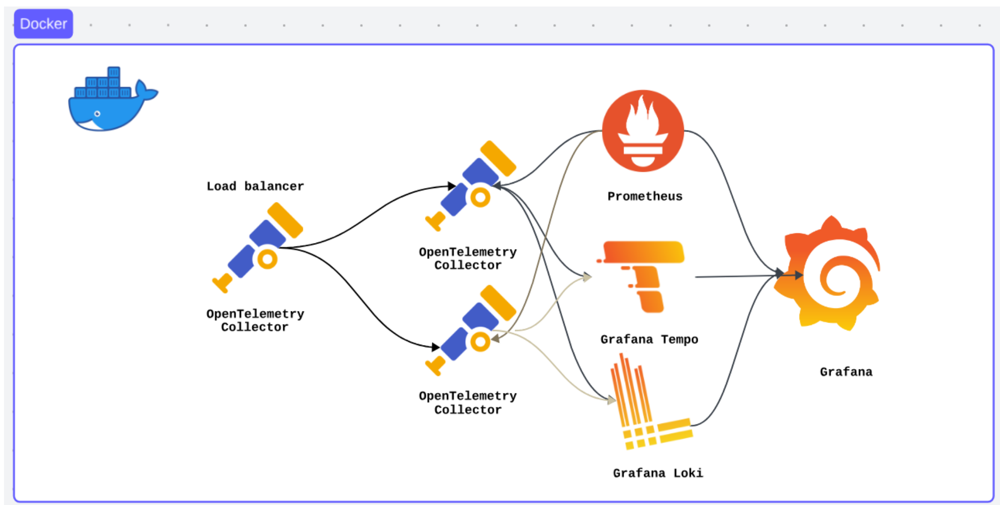
Suponiendo que ya no son solo dos aplicaciones las cuales envían telemetría, sino muchas más y, además, están siendo utilizadas por múltiples usuarios al mismo tiempo, el sistema de observabilidad podría tener dificultades para soportar todo el tráfico recibido.

Para aliviar el tráfico y evitar que dicha problemática ocurra constantemente, se puede implementar un Balanceador de Carga que se caracteriza por distribuir la información que llega a diferentes servicios con el objetivo de aumentar el rendimiento del aplicativo. Existen varios tipos de balanceadores de carga, pero en este caso se usa un balanceador de carga de capa siete debido a que se necesita manipular el sistema de comunicación que se utiliza.

Nativamente OpenTelemetry no se considera como un balanceador de carga, sin embargo, tiene la capacidad de funcionar como uno gracias a que su comunidad ha implementado una réplica de la imagen del Collector donde contemplaron esta funcionalidad, por tanto, se utiliza esta herramienta como balanceador de cargas. También, se duplica la instacia del Broker para así contar con dos (2) a los cuales poder repartir las cargas.

Figura 29.

Gráfico de la arquitectura del escenario 6 de la implementación



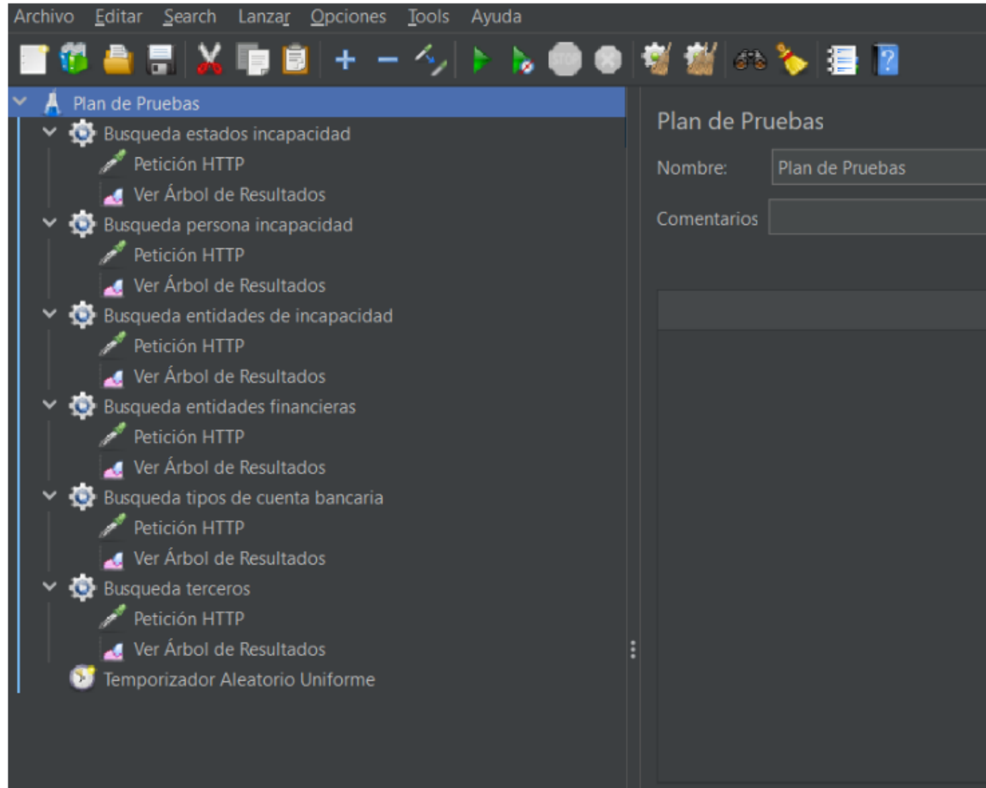
5.3. Pruebas

5.3.1. Simulación de múltiples usuarios

Retomando el escenario 6 de la implementación, en el entorno desarrollado se simulan múltiples peticiones a través de JMeter. Creamos un plan de pruebas con diferentes hilos, donde cada hilo contiene peticiones HTTP a nuestras aplicaciones de SpringBoot (recordemos que los servicios utilizados son Contratación y Situaciones administrativas). Cada petición se realiza múltiples veces, entonces también agregamos un temporizador para aumentar los tiempos entre peticiones.

Figura 30.

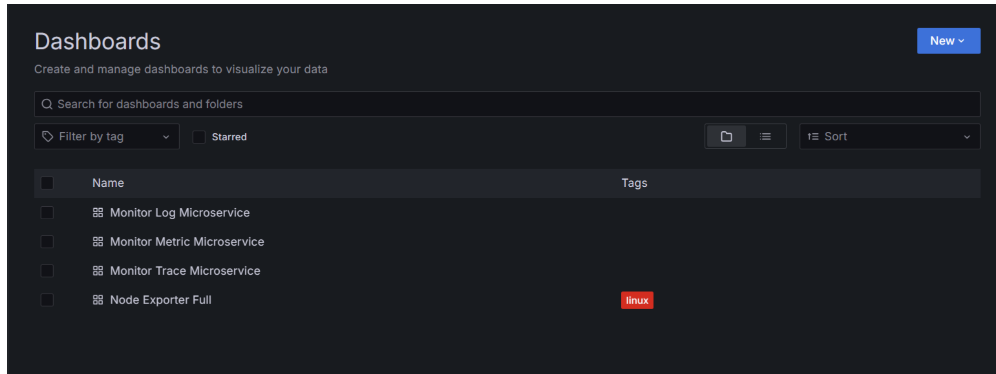
Gráfico del plan de pruebas realizado en JMeter



Para cada aplicación monitoreada, se puede identificar su telemetría a través de sus labels. Dependiendo de sus requerimientos, algunos servicios contarán con datos o telemetría exportada que otros no tendrán. En este escenario de pruebas se diseñaron 3 dashboards para visualizar logs, trazas y métricas, donde se incluye una variable que tomará el scope del servicio al que se le proyectarán los datos en Grafana.

Figura 31.

Gráfico de los dashboards creados para visualizar los datos de las aplicaciones de SpringBoot



A continuación presentamos los datos exportados:

- Logs

Figura 32.

Gráfico de los logs de la aplicación de Contratación después de ejecutar el plan de pruebas de JMeter

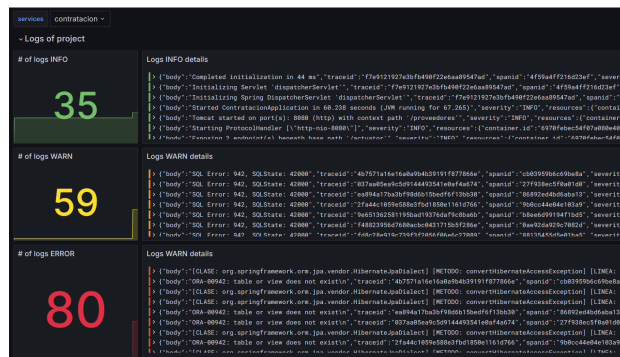
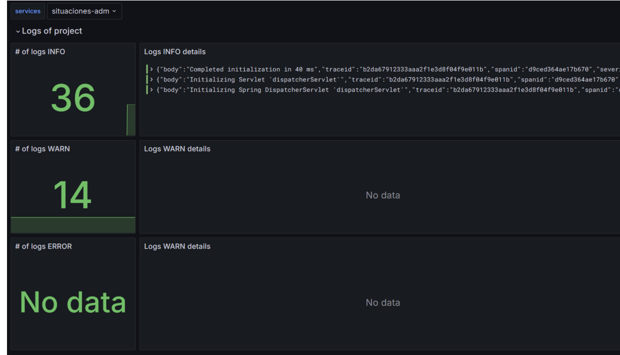


Figura 33.

Gráfico de los logs de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter



- Trazas

Figura 34.

Gráfico de las trazas de la aplicación de Contratación después de ejecutar el plan de pruebas de JMeter

services | contratacion

Traces of project

Trace

Trace ID	Start time	Service	Name	Duration
e396b3811606cc98b6c...	2024-04-07 14:52:52	<root span not yet received>		95 ms
1a895ff800bdf91eca0b...	2024-04-07 14:52:52	<root span not yet received>		92 ms
2264708411aa31a5384...	2024-04-07 14:52:51	contratacion	GET /proveedores/api/vTipoCuentaBancaria/	799 ms
9b01956df548666576c...	2024-04-07 14:52:50	contratacion	GET /proveedores/api/entidadFinanciera/sug	1.56 s
164c77fa7b5c0326da3...	2024-04-07 14:52:50	contratacion	GET /proveedores/api/entidadFinanciera/sug	359 ms
37aa05ea9c5d914449...	2024-04-07 14:52:49	contratacion	GET /proveedores/api/terceros/ent/pageable	2.54 s
239c5f45c47c95c9621...	2024-04-07 14:52:49	contratacion	GET /proveedores/api/entidadFinanciera/sug	2.64 s
1fa52cbf2c3c911a5eca...	2024-04-07 14:52:48	contratacion	GET /proveedores/api/entidadFinanciera/sug	2.86 s
1e88295d96d1385f545...	2024-04-07 14:52:48	contratacion	GET /proveedores/api/terceros/ent/pageable	2.85 s
6e7255021bc5f726857...	2024-04-07 14:52:48	contratacion	GET /proveedores/api/terceros/ent/pageable	1.95 s
19bccdf3021e4a141956...	2024-04-07 14:52:48	contratacion	GET /proveedores/api/vTipoCuentaBancaria/	3.08 s
1da4d2f16a154968650...	2024-04-07 14:52:48	contratacion	GET /proveedores/api/terceros/ent/pageable	1.74 s
1fc2980af92c1d67faa4...	2024-04-07 14:52:47	contratacion	GET /proveedores/api/terceros/ent/pageable	2.89 s

Figura 35.

Gráfico de las trazas de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter

services | situaciones-adm

Traces of project

Trace

Trace ID	Start time	Service	Name	Duration
21e436205fd4bc172ed...	2024-04-07 14:52:50.244	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	224 ms
571b25f4c6456485e6b...	2024-04-07 14:52:49.261	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	1.39 s
2fe92ba5320ccd9966...	2024-04-07 14:52:48.986	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	1.55 s
44899f2821c9cc86c39...	2024-04-07 14:52:48.979	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	1.57 s
36baf91c2182ac15a4f3...	2024-04-07 14:52:48.927	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	1.59 s
1a82570cdffb8b37f310...	2024-04-07 14:52:48.875	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	1.28 s
521c97d71484850f71ac...	2024-04-07 14:52:48.559	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	1.60 s
42fbb6ff131fddcb6865...	2024-04-07 14:52:48.325	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	1.92 s
410fa4e1090503e61a6...	2024-04-07 14:52:48.299	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	1.69 s
439e0b8d4a4a85efaac...	2024-04-07 14:52:48.075	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	1.97 s
2420b6492cb8884499...	2024-04-07 14:52:48.071	situaciones-adm	GET /situacionesadm/api/vEstadoIncapacida	2.05 s
33a88e2c58bc4960cd...	2024-04-07 14:52:47.733	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	2.58 s
4dfc6a7dc370a8057a6...	2024-04-07 14:52:47.624	situaciones-adm	GET /situacionesadm/api/entidadesIncapaci	2.65 s

- Métricas

Figura 36.

Gráfico de las métricas de la aplicación de Contratación después de ejecutar el plan de pruebas de JMeter

Figura 37.

Gráfico de las métricas de la aplicación de Situaciones Administrativas después de ejecutar el plan de pruebas de JMeter

5.3.2. Servicios no disponibles

Simulamos que nuestras herramientas de monitoreo no se encuentran disponibles, entonces procedemos a apagarlas.

Figura 38.

Gráfico de los servicios activos en nuestro sistema de observabilidad

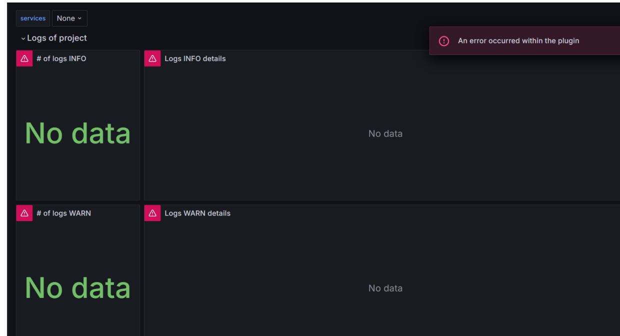
```
CONTAINER ID   NAME                              CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
       BLOCK I/O       PIDS
4e0372c99fbb   grafana                           6.03%     58.22MiB / 1.918GiB   2.96%     6.49MB / 22
MB     37MB / 2.26MB   11
3bba9e710151   proyecto-grado-loadBalancer-1     0.32%     40.14MiB / 1.918GiB   2.04%     7.47MB / 1.
55MB   3.67MB / 0B     9
5b0d4d6b00ed   node_exporter                     0.00%     8.031MiB / 1.918GiB   0.41%     154kB / 2.8
6MB    0B / 0B         5
32df6aa28ae9   proyecto-grado-collector-1        0.00%     24.92MiB / 1.918GiB   1.27%     354kB / 365
kB     1.2MB / 0B      8
9d9469eaa60f   proyecto-grado-collector2-1       0.00%     30.98MiB / 1.918GiB   1.58%     1.3MB / 1.3
6MB    3.76MB / 0B     9
```

Nota: Los servicios Prometheus, Tempo y Loki no se ven en la figura debido a que se encuentran apagados.

La caída de estas aplicaciones ocasiona que la telemetría no pueda ser transportada hasta la herramienta de visualización. Por tanto, al intentar ver los dashboards en Grafana aparecen errores debido a que no detecta las datasources.

Figura 39.

Gráfico error en Grafana al no tener datasources activas



Hacemos uso nuevamente del plan de pruebas de JMeter implementado en Simulación de múltiples usuarios. Después de un tiempo encendemos los servicios apagados en el sistema de observabilidad y nos dirigimos a los dashboards.

Figura 40.

Gráfico datos exportados cuando las herramientas de monitoreo se encontraban abajo

Nota: Vemos en este dashboard de métricas que en el intervalo de tiempo de 15:33 a 15:50 no existe información debido a que en ese periodo las herramientas Ptometheus, Tempo y Loki no se encontraban disponibles.

Como resultado nuestra instancia de OpenTelemetry Collector que reciba los datos de las aplicaciones de SpringBoot deberá retener estos hasta que los servicios vuelvan a estar arriba, por tanto cualquier petición realizada cuando las herramientas no se encuentran disponibles deben mostrarse una vez estén funcionando otra vez.

5.3.3. Evaluando el balanceador de cargas

También se realizan pruebas sobre el balanceador de cargas para determinar si funciona correctamente ante cualquier eventualidad. Para empezar, apagamos uno de los dos servicios Broker. En este caso el servicio proyecto-grado-collector-1 se encuentra apagado.

Figura 41.

Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-grado-collector-1 se encuentra apagado

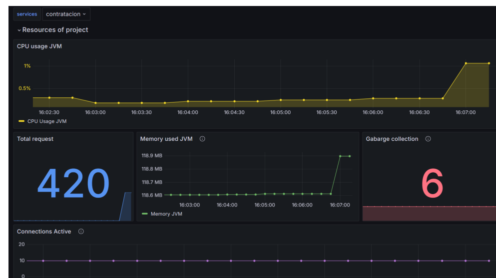
```
CONTAINER ID   NAME                          CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
       BLOCK I/O         PIDS
4e0372c99fbb   grafana                       0.17%     66.89MiB / 1.918GiB   3.41%     7.58MB / 23
.5MB   38.2MB / 2.48MB   11
8a71475c6bb1   tempo                         5.88%     72.21MiB / 1.918GiB   3.68%     40kB / 26.6
kB     5.62MB / 836kB    8
3bba9e710151   proyecto-grado-loadBalancer-1 0.52%     45.13MiB / 1.918GiB   2.30%     13.6MB / 2.
5MB    6.11MB / 0B       9
2db151675f31   loki                          0.66%     44.83MiB / 1.918GiB   2.28%     118kB / 365
kB     2.69MB / 1.12MB   9
f6f07ab1665f   prometheus                    0.00%     45.91MiB / 1.918GiB   2.34%     1.05MB / 19
9kB    5.56MB / 2.09MB   8
5b0d4d6b00ed   node_exporter                 0.00%     8.258MiB / 1.918GiB   0.42%     189kB / 3.5
2MB    0B / 0B           5
9d9469eaa60f   proyecto-grado-collector2-1   0.00%     35.35MiB / 1.918GiB   1.80%     2.05MB / 1.
76MB   5.07MB / 0B       9
```

Nota: El servicio proyecto-grado-collector-1 no se ve en la figura debido a que se encuentra apagado.

El balanceador de cargas recibe los datos provenientes del aplicativo y determina a cuál de los posibles destinos deberá reenviarlos. Para comprobar que el balanceador de cargas funciona según lo esperado, se envían peticiones al sistema de observabilidad, el cual recibe las peticiones y las expone en el Dashboard.

Figura 42.

Gráfico de los datos exportados cuando proyecto-grado-collector-1 se encuentra apagado



Ahora, se vuelve a encender el servicio Broker y se desactiva el otro. Se enciende nuevamente el servicio proyecto-grado-collector-1 y se desactiva el servicio proyecto-grado-collector2-1.

Figura 43.

Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-grado-collector2-1 se encuentra apagado

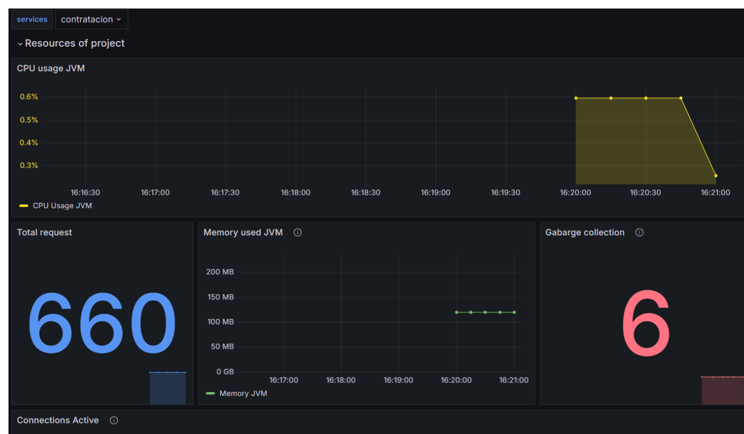
```
CONTAINER ID   NAME                              CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
       BLOCK I/O         PIDS
4e0372c99fbb   grafana                           0.25%     61.01MiB / 1.918GiB   3.11%     7.9MB / 23.
9MB    38.2MB / 2.52MB   12
8a71475c6bb1   tempo                             0.76%     51.34MiB / 1.918GiB   2.61%     63.3kB / 90
.7kB   6.14MB / 1.96MB   8
3bba9e710151   proyecto-grado-loadBalancer-1     0.70%     46.32MiB / 1.918GiB   2.36%     14MB / 2.59
MB     6.63MB / 0B       9
2db151675f31   loki                              0.94%     42.86MiB / 1.918GiB   2.18%     119kB / 365
kB     2.69MB / 1.38MB   9
f6f07ab1665f   prometheus                        0.04%     37.83MiB / 1.918GiB   1.93%     1.48MB / 30
2kB    5.62MB / 3.28MB   9
5b0d4d6b00ed   node_exporter                     0.00%     8.285MiB / 1.918GiB   0.42%     201kB / 3.7
3MB    0B / 0B           5
32df6aa28ae9   proyecto-grado-collector-1        0.00%     30.43MiB / 1.918GiB   1.55%     2.29kB / 1.
27kB   14.3MB / 0B       6
```

Nota: El servicio proyecto-grado-collector2-1 no se ve en la figura debido a que se encuentra apagado.

Nuevamente se comprueba que el balanceador de cargas funciona correctamente al apreciar que la telemetría se sigue enviando sin problemas.

Figura 44.

Gráfico de los datos exportados cuando proyecto-grado-collector2-1 se encuentra apagado



Finalmente, como última prueba, se simulan ambos servicios inactivos, dejando solo al balanceador de cargas sin poder conectarse a los servicios backend de monitoreo.

Figura 45.

Gráfico de las herramientas del sistema de observabilidad y su estado. proyecto-grado-collector-1 y proyecto-grado-collector2-1 se encuentran apagados

```
CONTAINER ID   NAME                            CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
       BLOCK I/O         PIDS
4e0372c99fbb   grafana                         0.26%     75.43MiB / 1.918GiB   3.84%     9.15MB / 29
.2MB   45.9MB / 2.8MB    12
8a71475c6bb1   tempo                           0.66%     64.54MiB / 1.918GiB   3.29%     153kB / 305
kB     7.24MB / 5.78MB   8
3bba9e710151   proyecto-grado-loadBalancer-1   0.61%     45.14MiB / 1.918GiB   2.30%     15.2MB / 2.
78MB   6.77MB / 0B       9
2db151675f31   loki                            0.55%     45.57MiB / 1.918GiB   2.32%     122kB / 367
kB     2.7MB / 2.27MB    9
f6f07ab1665f   prometheus                      0.37%     47.16MiB / 1.918GiB   2.40%     2.65MB / 64
5kB    7.06MB / 6MB      9
5b0d4d6b00ed   node_exporter                   1.92%     8.34MiB / 1.918GiB    0.42%     230kB / 4.2
9MB    0B / 0B           5
```

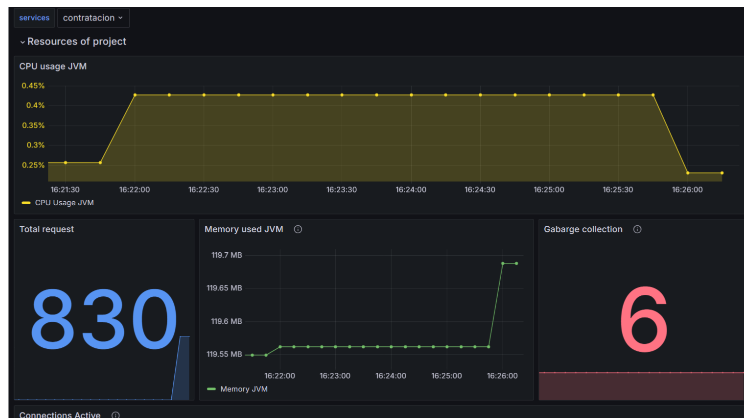
Nota: Los servicios proyecto-grado-collector-1 y proyecto-grado-collector2-1 no se ven en la figura debido a que se encuentran apagados.

Se realiza nuevamente el primer escenario de pruebas Simulación de múltiples usuarios, y después de un tiempo se activan nuevamente los servicios de proyecto-grado-collector-1 y proyecto-grado-collector2-1.

Toda la información que recibió el balanceador de cargas queda almacenada hasta que existan servicios a los cuales enviarla. En el dashboard de Grafana se puede ver la telemetría exportada durante el tiempo que los servicios proyecto-grado-collector-1 y proyecto-grado-collector2-1 se encontraban fuera de servicio.

Figura 46.

Gráfico de los datos exportados cuando proyecto-grado-collector-1 y proyecto-grado-collector2-1 se encuentran apagados



5.4. Evaluación y análisis de resultados

Cada fase realizada en el desarrollo es requisito para avanzar al paso siguiente. Al analizar la conceptualización de observabilidad y diseñar la arquitectura, se logran obtener los "planos"de la herramienta de observabilidad, los cuales serán utilizados en la implementación; en la implementación se pasa de la teoría a la práctica. La división de escenarios permite comprender cada implementación realizada en nuestro sistema de observabilidad; finalmente, las pruebas verifican la calidad del prototipo diseñado antes de que se ponga a disposición de los usuarios finales, y determinan si el sistema de observabilidad cumple con las funcionalidades esperadas y las realiza de manera efectiva.

Teniendo en cuenta los objetivos del proyecto, se obtienen algunas conclusiones del escenario 6 de la implementación:

- El prototipo logra recibir la telemetría exportada que incluye Logs, Métricas y Trazas, aún cuando múltiples usuarios realizan peticiones en periodos de tiempos muy cortos, esto satisface el objetivo general del proyecto.
- Si bien es cierto que se muestran los datos en nuestra herramienta de visualización, estos llegan cierta latencia, haciendo que los tiempos entre el llamado de las peticiones y la visualización no sean inmediatos.
- Al no tener disponibilidad de algunos servicios en nuestro sistema de observabilidad, las instancias de OpenTelemetry Collector (balanceador de cargas y brokers) retienen los datos de las aplicaciones por un periodo de tiempo hasta que nuevamente todas las herramientas se encuentren disponibles y el sistema esté funcional para posteriormente exportarlos y poder consultarlos en el visualizador

6. Conclusiones

En el transcurso de este trabajo, se ha sumergido sobre los diferentes conceptos que abarca la filosofía de la observabilidad, estudiando sistemas de arquitectura para monitorear microservicios, tanto como ejemplos, como aplicaciones ya en producción; diseñando un modelo arquitectónico usado para estudiar su evolución, adaptabilidad a las distintas pruebas realizadas y finalmente exponerlo en casos reales de proyectos de desarrollo como R.S.I. Por otro lado, el informe tiene la capacidad de responder las preguntas formuladas inicialmente[3], se han visualizado distintos factores que pueden llegar a perjudicar la arquitectura, sin embargo, también se han propuesto soluciones a dichas dificultades, por otro lado, a medida que se presentaban los distintos escenarios, se presenciaba como el modelo de observabilidad presentaba una mayor carga que se refleja en un mayor consumo de tiempo, es por ello, que es necesario recurrir a más instancias del broker e implementar un balanceador de cargas.

Se ha hecho un basto recorrido en conocer las definiciones de observabilidad, los entes que la componen, sobre como funciona realmente, entender acerca los tres pilares fundamentales de la observabilidad y de que se encargan cada uno de ellos, así como comprender la información que presentan, la importancia de implementar un sistema de monitoreo en un microservicio y diferenciar monitoreo con observabilidad, todo esto con el objetivo de poder tener un marco de referencia más amplio que permite establecer una série de parámetros a tener en cuenta sobre el desarrollo del primer diseño de arquitectura.

En cuanto a la implementación, se evidencia en cada escenario como evoluciona el diseño arquitectónico del sistema de observabilidad, adaptando nuevas estrategias que se traducen en una mejor eficiencia y eficacia. Esto ayuda a evitar problemas de sobrecarga y latencia durante las pruebas, y permite mantener la información del estado de los

---

[3] Ver capítulo 1

microservicios incluso después de encender ambos recolectores de datos tras haber sido apagados.

Finalmente, se puede asegurar la creación de un prototipo de sistema de observabilidad confiable, el cual mantiene a los desarrolladores informados sobre el estado de los aplicativos, cuenta con una arquitectura adaptable que permite mejoras y se ajusta a cualquier escenario que se presente.

7. Trabajo Futuro

El enfoque del prototipo se centra en los entornos backends de los servicios, lo cual abre las puertas para avanzar hacia arquitecturas más sofisticadas y complejas que también abarquen la parte frontend en futuros proyectos. Además, es crucial considerar que la observabilidad incluye la parte de testing; es decir, la capacidad de integrar mejoras como la automatización de pruebas al ensayar historias de usuario ya finalizadas. Esta automatización no solo ahorra tiempo, sino que también permite al equipo detectar fallos que, de haberse realizado manualmente, podrían haberse pasado por alto.

Una de las pruebas que se tenía planeado realizar era haber implementado el prototipo en un clúster de K3s otorgado por la R.S.I, el objetivo sería estudiar que tan factible resulta ser el diseño de la última arquitectura propuesta en un ambiente mucho más realista como lo es dicho proyecto.

Este prototipo utiliza herramientas Open Source y fáciles de implementar. De ser necesario por el proyecto de R.S.I., se pueden reemplazar las herramientas de monitoreo por las alternativas deseadas (si se desea continuar con OpenTelemetry es importante leer en la documentación si la alternativa a implementar contiene soporte oficial). La implementación de Kubernetes al sistema de observabilidad facilitaría el proceso de orquestación de servicios y la implementación del balanceador de cargas. Tener en cuenta que OpenTelemetry posee soporte oficial para Kubernetes.

Bibliografía

Amazon Web Services. (s.f.-a). *How Amazon CloudWatch works*. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_architecture.html

Amazon Web Services. (s.f.-b). *Observability vs Monitoring*. https://aws.amazon.com/es/compare/the-difference-between-monitoring-and-observability/#:~:text=Monitoring%20is%20the%20process%20of,the%20root%20cause%20of%20issues

Amazon Web Services. (2018, junio). *Monitorización y observabilidad de Amazon: Amazon CloudWatch - AWS*. https://aws.amazon.com/es/cloudwatch/

Apache SkyWalking. (2017). *Apache SkyWalking*. https://skywalking.apache.org/

de Vries, Sjouke and Blaauw, Frank and Andrikopoulos, Vasilios. (2023). Cost-Profiling Microservice Applications Using an APM Stack. *Future Internet*, *15*(1). https://doi.org/10.3390/fi15010037

Elastic. (2013). *What is Elastic Observability?* https://www.elastic.co/guide/en/observability/current/observability-introduction.html

Grafana Labs. (s.f.-a). *Get started with Grafana Tempo*. https://grafana.com/docs/tempo/latest/getting-started/

Grafana Labs. (s.f.-b). *Loki overview*. https://grafana.com/docs/loki/latest/get-started/overview/

Grafana Labs. (2014). *Technical documentation*. https://grafana.com/docs/

IBM. (2021, febrero). *IBM Instana Observability*. https://www.ibm.com/es-es/products/instana

LAUDON, KENNETH C. and LAUDON, JANE P. (2016). *SISTEMAS DE INFORMACION GERENCIAL - Decimocuarta edición*. PEARSON EDUCACIÓN.

Microsoft Azure. (2017, marzo). *Azure Monitor: herramientas modernas de observabilidad*. https://learn.microsoft.com/es-es/azure/azure-monitor/overview

Monasca. (2015). *Monasca*. https://wiki.openstack.org/wiki/Monasca

OpenTelemetry. (2022, octubre). *What is OpenTelemetry*. https://opentelemetry.io/docs/what-is-opentelemetry/

OpenTelemetry. (2024, enero). *Documentation*. https://opentelemetry.io/docs/

Prometheus. (2014, noviembre). *OVERVIEW*. https://prometheus.io/docs/introduction/overview/

Red Hat. (2023a). *¿Qué es una arquitectura de aplicaciones?* https://www.redhat.com/es/topics/cloud-native-apps/what-is-an-application-architecture

Red Hat. (2023b). *El concepto de DevOps*. https://www.redhat.com/es/topics/devops

Richardson, C. (2023). *What are microservices?* https://microservices.io/

Splunk. (2004). *Welcome to Splunk Observability Cloud*. https://docs.splunk.com/observability/en/get-started/welcome.html

Usman, Muhammad and Ferlin, Simone and Brunstrom, Anna and Taheri, Javid. (2022). A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access*, *10*, 86904-86919. https://doi.org/10.1109/ACCESS.2022.3193102

Apéndices

1. Repositorio que contiene el sistema de observabilidad

https://github.com/Camilo190/proyecto-grado

2. Tabla de las métricas para seleccionar herramientas de logs

Selección de herramientas para monitoreo de logs								
Peso: 1=Baja, 3=Media, 5=Alta Importancia			Rango	1	2	5	3	4
Nota : 1=Débil, 3=Promedio, 5=Se adecua fuertemente			Total	184	153	128	152	138
Familia/ Criterio			Peso	Otel	Loki	Logstash	Fluentd	Graylog
1	Funcionalidad							
	1,01	**Confiabilidad:** Capacidad de la herramienta de mantener su nivel de ejecución bajo condiciones normales en un período de tiempo establecido	4	5	4	3	4	4
	1,02	**Seguridad**: Habilidad de prevenir el acceso no autorizado, sea accidental o premeditado, a los programas y datos.	2	4	3	3	3	4
	1,03	**Alcance:** Consumo de recursos para soportar las cargas de trabajo.	5	5	4	3	3	3
	1,04	**Escalabilidad:** Capacidad de la herramienta para manejar un aumento en la carga de trabajo.	5	5	4	3	4	3
	1,05	**Rendimiento:** Capacidad de la herramienta bajo carga pesada en términos de rendimiento.	5	5	4	3	4	3
2	Usabilidad							
	2,01	**Fácilidad:** La facilidad de configuración e desarrollo de la herramienta.	3	3	4	3	4	3
	2,02	**Documentación:** Documentación clara de la herramienta que facilite su implementación y uso	3	4	4	3	4	3
3	Eficiencia							
	3,01	**Tiempo de implementación:** Tiempo que llevará implementar la herramienta y ponerla en producción.	1	3	3	3	2	3
	3,02	**Tiempo de respuesta:** Tiempo que tarda la herramienta en procesar y responder una solicitud.	4	4	3	3	3	3
4	Portabilidad y Mantenimiento							
	4,01	**Facilidad de mantenimiento:** Es fácil de mantener y actualizar.	4	4	3	3	3	4
5	Perspectiva de uso							
	5,01	**Comunidad y soporte:** Existe comunidad activa en torno a la herramienta y hay soporte del proveedor de la herramienta.	5	5	4	4	5	4
			Total	184	153	128	152	138

3. Tabla de las métricas para seleccionar herramientas de métricas

Selección de herramientas para monitoreo de metricas							
Peso: 1=Baja, 3=Media, 5=Alta Importancia			Rango	1	2	3	4
Nota : 1=Débil, 3=Promedio, 5=Se adecua fuertemente			Total	184	172	160	130
Familia/ Criterio			Peso	Otel	Prometheus	Elastic	Graphite
1	Funcionalidad						
	1,01	**Confiabilidad:** Capacidad de la herramienta de mantener su nivel de ejecución bajo condiciones normales en un período de tiempo establecido	4	5	4	5	3
	1,02	**Seguridad**: Habilidad de prevenir el acceso no autorizado, sea accidental o premeditado, a los programas y datos.	2	4	4	4	2
	1,03	**Alcance:** Consumo de recursos para soportar las cargas de trabajo.	5	5	4	4	3
	1,04	**Escalabilidad:** Capacidad de la herramienta para manejar un aumento en la carga de trabajo.	5	5	4	4	3
	1,05	**Rendimiento:** Capacidad de la herramienta bajo carga pesada en términos de rendimiento.	5	5	4	4	3
2	Usabilidad						
	2,01	**Fácilidad:** La facilidad de configuración e desarrollo de la herramienta.	3	3	4	3	3
	2,02	**Documentación:** Documentación clara de la herramienta que facilite su implementación y uso	3	4	5	4	3
3	Eficiencia						
	3,01	**Tiempo de implementación:** Tiempo que llevará implementar la herramienta y ponerla en producción.	1	3	4	3	3
	3,02	**Tiempo de respuesta:** Tiempo que tarda la herramienta en procesar y responder una solicitud.	4	4	4	4	4
4	Portabilidad y Mantenimiento						
	4,01	**Facilidad de mantenimiento:** Es fácil de mantener y actualizar.	4	4	4	3	3
5	Perspectiva de uso						
	5,01	**Comunidad y soporte:** Existe comunidad activa en torno a la herramienta y hay soporte del proveedor de la herramienta.	5	5	5	4	4
			Total	184	172	160	130

4. Tabla de las métricas para seleccionar herramientas de trazas

Selección de herramientas para monitoreo de trazas							
Peso: 1=Baja, 3=Media, 5=Alta Importancia			Rango	1	2	3	4
Nota : 1=Débil, 3=Promedio, 5=Se adecua fuertemente			Total	184	152	151	127
Familia/ Criterio			Peso	Otel	Tempo	Jaeger	Zipkin
1	Funcionalidad						
	1,01	**Confiabilidad:** Capacidad de la herramienta de mantener su nivel de ejecución bajo condiciones normales en un período de tiempo establecido	4	5	5	5	5
	1,02	**Seguridad**: Habilidad de prevenir el acceso no autorizado, sea accidental o premeditado, a los programas y datos.	2	4	3	3	2
	1,03	**Alcance:** Consumo de recursos para soportar las cargas de trabajo.	5	5	3	4	3
	1,04	**Escalabilidad:** Capacidad de la herramienta para manejar un aumento en la carga de trabajo.	5	5	4	4	3
	1,05	**Rendimiento:** Capacidad de la herramienta bajo carga pesada en términos de rendimiento.	5	5	4	4	3
2	Usabilidad						
	2,01	**Fácilidad:** La facilidad de configuración e desarrollo de la herramienta.	3	3	4	3	2
	2,02	**Documentación:** Documentación clara de la herramienta que facilite su implementación y uso	3	4	4	3	2
3	Eficiencia						
	3,01	**Tiempo de implementación:** Tiempo que llevará implementar la herramienta y ponerla en producción.	1	3	3	3	2
	3,02	**Tiempo de respuesta:** Tiempo que tarda la herramienta en procesar y responder una solicitud.	4	4	3	3	3
4	Portabilidad y Mantenimiento						
	4,01	**Facilidad de mantenimiento:** Es fácil de mantener y actualizar.	4	4	3	3	3
5	Perspectiva de uso						
	5,01	**Comunidad y soporte:** Existe comunidad activa en torno a la herramienta y hay soporte del proveedor de la herramienta.	5	5	4	4	4
			Total	184	152	151	127

5. Dockerfile en aplicación de SpringBoot de prueba (Java 17)

```
#
# BUILD
#
# Establecer la imagen base de Maven para compilar la aplicacion
FROM maven:3.8.3-openjdk-17 AS build
# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app
# Copiar el archivo pom.xml y descargar las dependencias de Maven
COPY pom.xml .
RUN mvn dependency:go-offline
# Copiar el codigo fuente y compilar la aplicacin
COPY src ./src
RUN mvn package -DskipTests


#
# PACKAGE
#
# Crear una nueva imagen basada en la imagen de Java 17
FROM eclipse-temurin:17-jdk-alpine


# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app
```

```
# Copiar el archivo JAR de la aplicacion desde la fase de compilacion
#al directorio de trabajo en el contenedor
COPY --from=build /app/target/my-spring-project-0.0.1-SNAPSHOT.jar .


# Comando para ejecutar la aplicacion cuando se inicie el contenedor
CMD ["java", "-jar", "my-spring-project-0.0.1-SNAPSHOT.jar"]
```

6. Enlace para descargar OpenTelemetry Java Agent

https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar

7. Dockerfile en aplicación de SpringBoot (Java 17) de prueba que ejecuta el OpenTelemetry Java Agent

```
#
# BUILD
#
# Establecer la imagen base de Maven para compilar la aplicacion
FROM maven:3.8.3-openjdk-17 AS build
# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app
# Copiar el archivo pom.xml y descargar las dependencias de Maven
COPY pom.xml .
RUN mvn dependency:go-offline
# Copiar el codigo fuente y compilar la aplicacion
COPY src ./src
```

```
RUN mvn package -DskipTests


#

# PACKAGE

#

# Crear una nueva imagen basada en la imagen de Java 17

FROM eclipse-temurin:17-jdk-alpine


# Establecer el directorio de trabajo dentro del contenedor

WORKDIR /app


# Copiar el archivo JAR de la aplicacion desde la fase de compilacin

#al directorio de trabajo en el contenedor

COPY --from=build /app/target/my-spring-project-0.0.1-SNAPSHOT.jar .


# Comando para ejecutar la aplicacion cuando se inicie el contenedor

COPY /opentelemetry-javaagent.jar opentelemetry-javaagent.jar

ENTRYPOINT java -jar -javaagent:opentelemetry-javaagent.jar my-spring-project-0.0.1-
    SNAPSHOT.jar
```

8. Dockerfile en aplicación de SpringBoot (Java 11) de R.S.I. que ejecuta el OpenTelemetry Java Agent

```
FROM adoptopenjdk:11-jre-hotspot

WORKDIR /spring-backend
```

```
COPY target/proveedores.war app.war

COPY /opentelemetry-javaagent.jar opentelemetry-javaagent.jar

ENTRYPOINT java -jar -javaagent:opentelemetry-javaagent.jar app.war
```

9. docker-compose en aplicación de SpringBoot de prueba (Java 17)

```
version: '3'

services:


  app:

    build: ./

    environment:

      OTEL_SERVICE_NAME: "agent-example-app"

      OTEL_EXPORTER_OTLP_ENDPOINT: "http://collector:4317"#URL de nuestro collector

      # Logs are disabled by default

      OTEL_LOGS_EXPORTER: "otlp"

      # Optional specify file configuration instead of using environment variable

    scheme

      # To use, call "export OTEL_CONFIG_FILE=/sdk-config.yaml" before calling docker

     compose up

      OTEL_CONFIG_FILE:

    ports:

      - "8080:8080"

    volumes:

      - ./sdk-config.yaml:/sdk-config.yaml
```

10. sdk-config de aplicación de SpringBoot de prueba (Java 17)

```
file_format: "0.1"


resource:

  attributes:

    service.name: agent-app


logger_provider:

  processors:

    - batch:

        exporter:

          otlp:

            endpoint: http://collector:4317

            protocol: grpc


tracer_provider:

  processors:

    - batch:

        exporter:

          otlp:

            endpoint: http://collector:4317

            protocol: grpc
```

```
meter_provider:
  readers:
    - periodic:
        exporter:
          otlp:
            endpoint: http://collector:4317
            protocol: grpc
  # Example of how to use view to drop a metric which isn't needed.
  # This configuration is not available with the environment variable configuration
    scheme.
  # NOTE: most users will want jvm.memory.limit metric and should remove this view.
  views:
    - selector:
        instrument_name: jvm.memory.limit
      stream:
        aggregation:
          drop:
```