eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Trabajo de Fin de Grado

Grado en Inteligencia Artificial

Ingeniería de Computadores

Infraestructura en la Nube y Detección de Anomalías: Mejoras y Despliegues con Kubernetes

Markel Ramiro Vaquero

Dirección
Jose Miguel Alonso

22 de abril de 2025

Agradecimientos

Resumen

Este trabajo aborda la reingeniería y mejora de una plataforma de venta y gestión de entradas, originalmente alojada en una instancia EC2 de AWS con recursos limitados y susceptibilidad a picos de demanda. Para mejorar su escalabilidad y eficiencia, la plataforma ha sido migrada a Kubernetes (K8s), aprovechando sus capacidades de orquestación automática y escalado dinámico.

La contenerización de la aplicación se ha realizado mediante Docker, seguida por el despliegue en un clúster gestionado en Google Kubernetes Engine (GKE). Esta transformación permitió manejar eficazmente las fluctuaciones en la demanda y sobre todo, mejorar la disponibilidad del servicio. Para la observabilidad, se ha implementado Prometheus para recopilar métricas del cliente de Flask, que se recolectan por defecto, facilitando la monitorización del tráfico, el rendimiento y la salud del sistema. Grafana se ha utilizado para visualizar estas métricas, creando paneles que permiten seguir en tiempo real el estado operativo de la plataforma.

Además, se desarrollaron técnicas de detección de anomalías basadas en el análisis estadístico de las métricas de Prometheus para identificar y alertar sobre comportamientos inusuales, como picos anómalos en la demanda o errores de aplicación, permitiendo intervenciones rápidas y efectivas. Los resultados ha demostrado que la plataforma no solo responde eficazmente a las exigencias de escalabilidad, sino que también ha mejorado su capacidad para anticiparse y resolver problemas operativos de manera proactiva. Este proyecto establece un precedente para la optimización de aplicaciones monolíticas en la nube y resalta la importancia de implementar observabilidad avanzada en la gestión eficiente de plataformas en entornos dinámicos.

Índice de contenidos

Índice de figuras

Índice de tablas

Lista de Algoritmos

CAPÍTULO 1

Introducción

El proyecto aborda los desafíos críticos enfrentados por una plataforma existente dedicada a la gestión y venta de entradas para eventos. Actualmente, la aplicación opera sobre una arquitectura monolítica alojada en un servidor EC2 de AWS. Aunque funcional, esta configuración ha demostrado ser insuficiente durante los períodos de alta demanda, característicos del sector de eventos, donde el tráfico de usuarios se intensifica drásticamente días antes de cada evento. Este incremento en la carga ha resultado en problemas recurrentes de rendimiento, como la saturación de la memoria del servidor, lo que lleva a fallos críticos del sistema y pérdidas de disponibilidad.

La arquitectura actual encapsula tanto el frontend como el backend en un único entorno de ejecución gestionado por Flask. Esto incluye la renderización de páginas web, la manipulación de hojas de estilo y scripts de JavaScript de gran tamaño (como Bootstrap), la gestión de la base de datos, la creación de entradas y códigos QR, el envío de correos electrónicos, y la API que facilita la comunicación entre el frontend y el backend. La concentración de todas estas funciones en un solo servicio no solo aumenta el consumo de memoria durante los picos de tráfico, sino que también incrementa el riesgo de interrupciones del servicio, ya que cualquier fallo afecta a toda la plataforma (*vid.* Figura 1.1).

Además, la ausencia de un sistema de alertas adecuado complica la detección oportuna de problemas, lo que a menudo resulta en tiempos de respuesta lentos para resolver fallos, afectando negativamente la experiencia del usuario y la imagen del negocio. La naturaleza crítica de estos problemas y su impacto directo en la operatividad y la rentabilidad de la plataforma subrayan la urgencia de una reingeniería profunda.

Por lo tanto, este proyecto propone una migración hacia una arquitectura más robusta y escalable utilizando un clúster de Kubernetes en la nube, lo cual permitirá una gestión más eficiente de los recursos y una escalabilidad automática según la demanda. Además, se planea integrar Prometheus y Grafana para implementar un sistema avanzado de observabilidad y alertas. Este enfoque no solo mejorará la capacidad de respuesta y la disponibilidad del sistema

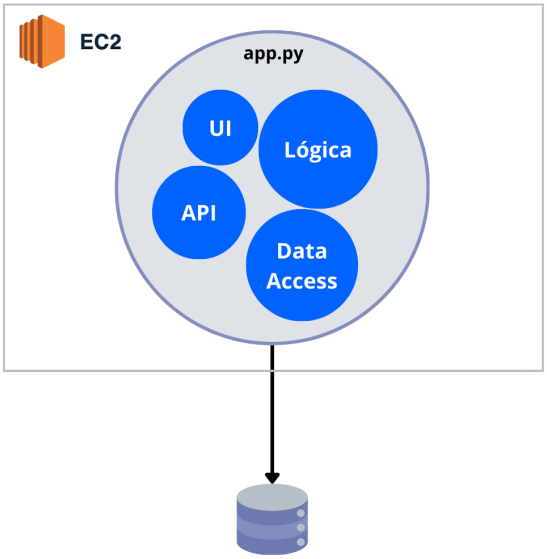

Figura 1.1: Arquitectura monolítica. Un servidor alojado en AWS, en el cual esta corriendo la aplicación. Fuera del servidor tenemos la Base de Datos.

durante los picos de demanda, sino que también facilitará una monitorización continua y proactiva del rendimiento de la aplicación.

Los objetivos propuestos son los siguientes:

1. Definir y optimizar la funcionalidad de la aplicación y sus componentes individuales para distribuir la carga de manera más efectiva.
2. Implementar una reingeniería de la aplicación para su despliegue en Kubernetes en la nube (AWS), permitiendo una escalabilidad dinámica y una gestión de recursos optimizada.
3. Integrar un sistema de telemetría utilizando Prometheus para la observación en tiempo real y la identificación temprana de problemas potenciales.
4. Desarrollar paneles de control en Grafana que permitan visualizar el estado operativo de la plataforma y activar alertas basadas en detección de anomalías sobre las métricas recolectadas.

Este proyecto busca no solo abordar las deficiencias técnicas actuales, sino también establecer una plataforma más resiliente y adaptable que pueda soportar el crecimiento futuro y mantener una alta satisfacción del cliente en un mercado competitivo.

CAPÍTULO 2

Descripción de la Aplicación

La aplicación que se describe en este trabajo de fin de grado es una plataforma de venta y gestión de entradas online, desarrollada previamente y actualmente en uso. Esta aplicación facilita la interacción entre organizadores de eventos y clientes, actuando como un punto de encuentro digital donde los eventos pueden ser gestionados y las entradas adquiridas.

2.1. Arquitectura General

La plataforma está implementada como una aplicación web monolítica y alojada en una instancia EC2 de Amazon Web Services (AWS). A continuación, se detallan los componentes principales de la aplicación:

- **Frontend:** Desarrollado con HTML, CSS y JavaScript, facilita la interacción del usuario, permitiendo a los organizadores gestionar eventos y a los clientes adquirir entradas.
- **Backend:** Implementado en Python usando el framework Flask, maneja la lógica de aplicación, procesamiento de datos y respuestas a las solicitudes de los usuarios.
- **Base de Datos:** Utiliza MySQL para almacenar datos relacionados con usuarios, organizadores de eventos, entradas y transacciones.

2.2. Componentes Clave

- **Sistema de Gestión de Usuarios:** Gestiona el registro, inicio de sesión y administración de perfiles de usuarios y organizadores.
- **Módulo de Gestión de Eventos:** Permite a los organizadores crear, editar y gestionar eventos, incluyendo precios y disponibilidad de entradas.

- **Módulo de Ventas:** Encargado de procesar las compras de entradas y generar tickets digitales para el acceso a eventos.

2.3. Funcionalidad y Flujos de Trabajo

La aplicación permite a los organizadores publicar detalles de eventos, y a los clientes navegar por estos eventos, seleccionar entradas y completar compras a través de un proceso de pago integrado. La arquitectura actual, aunque completamente operativa, presenta desafíos en términos de escalabilidad y mantenimiento debido a su naturaleza monolítica.

2.4. Plan de Reingeniería

Contrario a una transformación hacia una arquitectura de microservicios que implicaría la descomposición de la aplicación en servicios más pequeños e independientes, el enfoque de este TFG es añadir servicios de monitoreo, visualización y detección de anomalías. Estos servicios complementarán la aplicación monolítica existente sin modificar su estructura interna. El objetivo es mejorar la capacidad de observación y respuesta ante incidentes, así como manejar eficientemente los picos de carga sin reestructurar completamente la arquitectura actual de la aplicación.

Esta estrategia permite integrar mejoras significativas en términos de operabilidad y escalabilidad, manteniendo al mismo tiempo la integridad y la funcionalidad central de la plataforma original.

CAPÍTULO 3

Tecnologías Utilizadas

3.1. Flask

"Flask fue creado por Armin Ronacher de Pocoo, un grupo internacional de entusiastas de Python formado en 2004"[?]. Es un framework de desarrollo web escrito en Python que no requiere herramientas o bibliotecas particulares. En este proyecto, Flask fue elegido porque la aplicación original ya estaba desarrollada en Flask, lo cual facilitó su reutilización y adaptación. Además, Flask es ideal para nuestras necesidades actuales, ya que se integra muy bien con Prometheus, simplificando así la monitorización y facilitando el desarrollo del proyecto.

La plataforma de venta y gestión de entradas utiliza Flask para gestionar las interacciones con el usuario, el procesamiento de datos y la lógica de negocio. Su compatibilidad con contenedores Docker permite una fácil migración y despliegue en Kubernetes, mejorando la eficiencia y escalabilidad del sistema.

3.2. Prometheus

Prometheus es un sistema de monitoreo y alerta de código abierto, ampliamente utilizado en el ámbito de DevOps y la administración de sistemas. Fue desarrollado inicialmente en SoundCloud en 2012 y, desde entonces, ha sido adoptado por numerosas empresas y organizaciones, formando una comunidad activa de desarrolladores y usuarios. Como muestra de su independencia y estructura de gobernanza, Prometheus se unió a la Cloud Native Computing Foundation en 2016 como el segundo proyecto hospedado, después de Kubernetes.

3.2.1. Qué es una Métrica en Prometheus

En Prometheus, una métrica es un identificador que representa una medida cuantitativa de algún recurso o proceso en su sistema. Cada métrica está asociada con un conjunto de etiquetas (pares clave-valor) que proporcionan dimensiones adicionales para clasificar y filtrar.

Las métricas en Prometheus se almacenan como series temporales, donde cada punto en la serie representa el valor de la métrica en un momento específico, junto con su marca de tiempo correspondiente. Esto permite a Prometheus realizar un seguimiento preciso de cómo cambian las métricas a lo largo del tiempo.

3.2.2. Arquitectura de Prometheus

La arquitectura de Prometheus está diseñada para la recolección, almacenamiento y procesamiento eficiente de métricas desde diversas fuentes de una manera escalable. Los componentes clave de esta arquitectura incluyen (*vid.* Figura 3.1):

- **Servidor de Prometheus**: Compuesto por varios elementos esenciales:
  1. **Worker de Recolección de Datos**: Responsable de recolectar métricas de los objetivos, este componente extrae datos activamente siguiendo el modelo "pull". Las métricas son expuestas por los sistemas monitorizados en un endpoint específico, usualmente ‘/metrics‘, y Prometheus las recopila periódicamente.
  2. **Base de Datos de Series Temporales (TSDB)**: Optimizada para el almacenamiento y la gestión eficiente de métricas temporales, facilitando el acceso rápido y confiable a los datos históricos.
  3. **Servidor HTTP**: Permite la recuperación de los datos almacenados mediante consultas en su lenguaje específico, PromQL, facilitando la visualización y análisis de los datos.
- **Exportadores**: Facilitan la interacción entre el sistema de monitoreo y los sistemas objetivo extrayendo métricas y exponiéndolas en el formato esperado por Prometheus. Los exportadores son cruciales para sistemas que no exponen métricas directamente en un formato compatible con Prometheus.
- **Pushgateway**: Permite la integración de métricas de trabajos batch o de corta duración que no pueden ser recolectados de manera regular.
- **Gestor de Alertas (Alertmanager)**: Maneja alertas generadas por Prometheus, procesando y enroutando las notificaciones hacia los canales adecuados como correos electrónicos o sistemas de tickets.
- **Descubrimiento de Servicios**: Automatiza el descubrimiento de objetivos a monitorizar, reduciendo la necesidad de configuraciones estáticas y manuales.

En nuestro proyecto, se utiliza un exportador específico para recoger métricas generadas automáticamente por la aplicación Flask, como solicitudes HTTP, latencias y errores. Además, más adelante, en la sección de detección de anomalías, discutiremos cómo se emplea un exportador adicional para crear y visualizar una métrica de interés específico, aunque los detalles se expondrán más adelante en la sección pertinente.

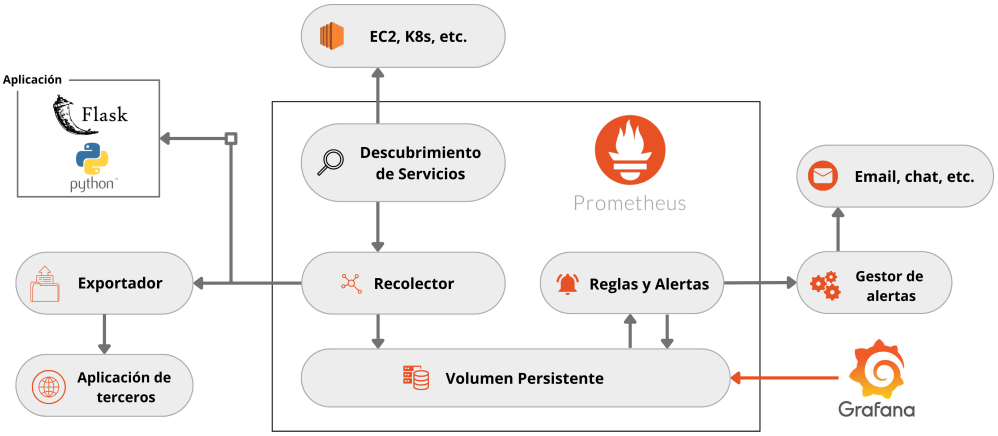

Figura 3.1: Esquema detallado de la arquitectura de Prometheus, mostrando sus componentes principales y el flujo de datos.

3.2.3. Uso de Prometheus en la Detección de Anomalías

Prometheus ha sido integrado eficazmente con Flask, aprovechando su capacidad para recolectar automáticamente métricas del cliente de Flask, lo que proporciona una visión clara y continua del rendimiento de la aplicación en tiempo real. Esto permite detectar y responder proactivamente a problemas potenciales.

3.3. Grafana

Grafana es una plataforma avanzada de análisis y visualización de datos de código abierto, que se ha convertido en una herramienta indispensable para monitorear infraestructuras de TI y analizar métricas en tiempo real. Grafana permite a los usuarios crear paneles de control dinámicos y visualmente atractivos, lo que facilita la interpretación y el análisis de grandes volúmenes de datos provenientes de diversas fuentes.

Entre las características clave de Grafana se incluyen su capacidad para integrar múltiples fuentes de datos como Prometheus, Kubernetes, Elasticsearch, y Microsoft SQL Server, entre otros (*vid.* Figura 3.2) . Esto permite a los usuarios combinar datos de diversas plataformas para obtener una visión más completa y detallada del rendimiento y la salud de sus sistemas. Sin embargo, en el contexto de este proyecto, Grafana se utiliza exclusivamente para visualizar las métricas recolectadas por Prometheus, que extrae datos directamente de nuestra aplicación Flask.

Esta integración entre Prometheus y Grafana proporcionará una solución robusta y completa para la observabilidad de la infraestructura en Kubernetes, ofreciendo una visión clara del estado operativo y la capacidad de respuesta de la aplicación ante las fluctuaciones de la demanda en la plataforma.

Figura 3.2: Ejemplo ilustrativo de un dashboard de monitoreo avanzado en Grafana, mostrando diversas métricas y paneles integrados con sistemas externos como AWS y Kubernetes.

3.4. Locust

Locust es una herramienta de pruebas de carga de código abierto que simula millones de usuarios accediendo simultáneamente a una aplicación web. A diferencia de otras herramientas que requieren interfaces de usuario complicadas o configuraciones basadas en XML, Locust permite a los usuarios definir el comportamiento de las pruebas directamente mediante código Python simple y directo. Esto facilita una simulación más realista del comportamiento humano al interactuar con la aplicación web.

3.4.1. Características de Locust

- **Definición de Comportamiento en Código:** En Locust, todas las pruebas se describen usando código Python, eliminando la necesidad de interfaces de usuario engorrosas o archivos XML.
- **Distribuido y Escalable:** Locust soporta la ejecución de pruebas de carga distribuidas en múltiples máquinas. Esto le permite simular millones de usuarios simultáneos, ideal para probar aplicaciones en escenarios de alto tráfico como plataformas de venta de entradas.

En el contexto de este Trabajo de Fin de Grado, Locust ha sido utilizado para generar tanto tráfico constante como picos de tráfico. Esta capacidad es crucial para plataformas que experimentan grandes fluctuaciones en la demanda, permitiéndonos evaluar cómo la infraestructura de Kubernetes maneja cargas intensas y ajustar configuraciones para optimizar el rendimiento bajo condiciones de tráfico real.

3.4.2. Ejemplo de Código

Aquí se muestra un ejemplo del código utilizado para definir el comportamiento de un usuario en Locust. Este script simula a un usuario que realiza tareas comunes en el sitio web, como iniciar sesión y acceder a diferentes páginas [?].

```
from locust import HttpUser, between, task

class WebsiteUser(HttpUser):
    wait_time = between(5, 15)

    def on_start(self):
        # Simula un inicio de sesion del usuario
        self.client.post("/login", {
            "username": "test_user",
            "password": "secure_password"
        })

    @task
    def index(self):
        # Accede a la pagina de inicio y a los recursos estaticos
        self.client.get("/")
        self.client.get("/static/assets.js")

    @task
    def about(self):
        # Accede a la pagina 'About'
        self.client.get("/about/")
```

Listing 3.1: Código de ejemplo para definir un usuario en Locust.

Este script demuestra cómo Locust permite simular interacciones detalladas y realistas con una aplicación, proporcionando insights valiosos sobre la capacidad de respuesta y la estabilidad de la aplicación bajo condiciones de uso intensivo.

3.5. Docker

Docker es una plataforma de contenedorización que permite empaquetar una aplicación y sus dependencias en un contenedor virtual que puede correr en cualquier sistema operativo que soporte Docker. Esto asegura coherencia en los entornos de desarrollo, prueba y producción, eliminando el clásico problema "funciona en mi máquina". En nuestro proyecto, Docker se ha utilizado para contenerizar la aplicación Flask, facilitando su despliegue y escalabilidad en Kubernetes. Esta contenerización no solo mejora la eficiencia del desarrollo y despliegue sino también asegura que la aplicación funcione uniformemente en distintos entornos.

3.6. Kubernetes

Kubernetes es un sistema de orquestación de contenedores de código abierto que automatiza la implementación, el escalado y la gestión de aplicaciones contenerizadas. Se ejecuta sobre

un clúster de máquinas físicas o virtuales, proporcionando las herramientas necesarias para operar aplicaciones a escala y con alta disponibilidad. Utiliza una serie de abstracciones a nivel de sistema para facilitar la gestión de la infraestructura y las aplicaciones. A continuación, se describen los componentes clave y algunos conceptos fundamentales utilizados en nuestro proyecto:

- **Pods:** Son la unidad más pequeña que puede ser desplegada y gestionada por Kubernetes. Un pod representa una instancia de una aplicación en ejecución y puede contener uno o varios contenedores que comparten recursos.
- **Deployments:** Facilitan la gestión de múltiples réplicas de una aplicación, asegurando que el número deseado de instancias esté siempre funcionando y maneje adecuadamente las actualizaciones automáticas y los *rollbacks* (reversión a versiones anteriores en caso de fallo durante una actualización).
- **ConfigMaps:** Son objetos en Kubernetes que permiten almacenar información de configuración no confidencial, como archivos de configuración, argumentos de comando y variables de entorno, en un formato clave-valor. Esto permite desacoplar la configuración específica del entorno de las imágenes del contenedor, facilitando la gestión de aplicaciones sin la necesidad de reconstruir las imágenes [**?**].
- **Services:** Proporcionan un mecanismo para exponer aplicaciones que se ejecutan en un conjunto de Pods como un servicio de red. Los *ClusterIP*, que son el tipo de servicio por defecto, permiten que los servicios sean alcanzables solo dentro del clúster, facilitando la comunicación interna y la gestión del tráfico entre diferentes componentes de la aplicación.
- **Ingress:** Como forma más avanzada de repartir la carga de trabajo entre diferentes pods, como complemento a los servicios, se puede usar un recurso denominado ingress. Con un ingress podemos repartir la carga de entrada entre diferentes servicios en función del URL empleado (*vid.* Figura 3.3) . También puede hacer de extremo en una comunicación TLS. Un ingress requiere que en el clúster esté instalado un controlador de ingreso.
- **Horizontal Pod Autoscaler (HPA):** Permite a Kubernetes ajustar automáticamente el número de pods en un deployment, basado en la observación de métricas como el uso de CPU o memoria. Es importante aclarar que aunque HPA puede utilizar métricas proporcionadas por Prometheus, su funcionamiento no depende de este sistema de monitoreo.

La implementación de estos componentes y la estructura de Kubernetes permiten a nuestro proyecto mejorar significativamente la escalabilidad, la disponibilidad y la gestión de los recursos de la aplicación, abordando efectivamente las necesidades de un entorno de producción dinámico.

Además de los componentes mencionados, la gestión y operación de un clúster de Kubernetes se realiza frecuentemente utilizando la herramienta de línea de comandos **kubectl**.

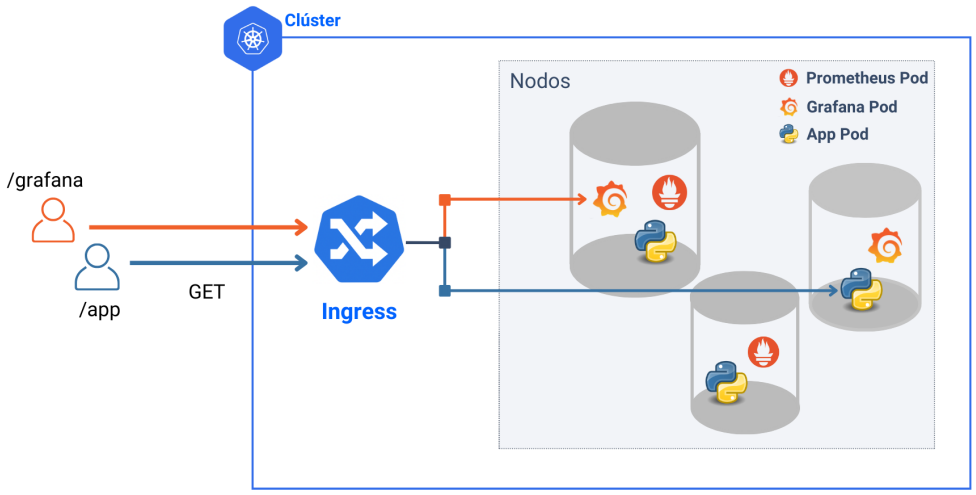

Figura 3.3: Uso de un recurso Ingress para, en función de una URL de una petición, redirigirla a Grafana o a la aplicación Flask

Esta herramienta es esencial para interactuar con el clúster, permitiendo a los desarrolladores y administradores ejecutar comandos contra los clústeres de Kubernetes para desplegar aplicaciones, inspeccionar y gestionar recursos del clúster y ver logs.

3.6.1. Manifiestos

Para el despliegue y configuración de recursos dentro de Kubernetes, se utilizan archivos de manifiesto en formato **YAML** (o **YML**). Estos archivos definen los recursos que necesitan ser creados y gestionados, como pods, deployments, services, y ConfigMaps. Una práctica común para simplificar la gestión de múltiples recursos es combinar varios de estos recursos en un único archivo *YAML*, separando cada definición de recurso con tres guiones (‘—‘). Esto permite a los usuarios crear o actualizar recursos de manera atomizada y coherente, facilitando el despliegue y la actualización de aplicaciones complejas con un solo comando, como se muestra en el siguiente ejemplo:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  ...
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
```

```
spec:
  ...
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  ...
```

Listing 3.2: Ejemplo de manifiesto que está compuesto por un deployment, service y configMap

Esta estructuración de archivos *YAML* no solo mejora la legibilidad y el mantenimiento del código de infraestructura, sino que también optimiza el proceso de despliegue automático en entornos de desarrollo, prueba y producción. La capacidad de gestionar y desplegar configuraciones complejas de forma eficiente es fundamental para mantener la agilidad y la capacidad de respuesta en entornos de producción dinámicos.

CAPÍTULO 4

Diseño de la Solución

4.1. Arquitectura

Esta sección estoy pensando en quitarla

4.2. Diseño de los Componentes

4.3. Diagramas

CAPÍTULO 5

Desarrollo y Pruebas

5.1. Proceso de Desarrollo

El desarrollo del proyecto comenzó estableciendo un entorno de pruebas local que pudiera simular las características de un entorno de producción, pero de una manera más controlada y accesible. Para esto, se optó por utilizar Docker Desktop, una solución que permite ejecutar contenedores Docker y orquestarlos mediante Kubernetes directamente en un sistema operativo Windows 10.

Docker Desktop integra de manera nativa con Windows 10 a través del Subsistema de Windows para Linux versión 2 (WSL2) [**?**]. WSL2 es una capa de compatibilidad innovadora diseñada por Microsoft para permitir una máquina virtual Linux integrada directamente en Windows 10. Esto es especialmente útil para poder utilizar herramientas de Linux en un entorno Windows, como es el caso de utilizar Kubernetes y Docker juntos.

Utilizando WSL2, Docker Desktop ofrece un clúster de Kubernetes que se ejecuta dentro de esta máquina virtual Linux. Este clúster se compone de un único nodo, lo que es suficiente para las fases iniciales del desarrollo y pruebas de aplicaciones que serán desplegadas utilizando Kubernetes. En este clúster, es posible desplegar "deploymentsz "services.[en]tre otros, esencial para simular cómo se comportará la aplicación en un entorno de producción (*vid.* Figura 5.1) .

5.2. Despliegue de Kubernetes

El despliegue en Kubernetes involucra la configuración y puesta en marcha de varios servicios clave que son fundamentales para el funcionamiento de nuestra plataforma de venta y gestión de entradas. Estos servicios son desplegados utilizando Kubernetes y se configuran a través de archivos YAML específicos para cada componente.

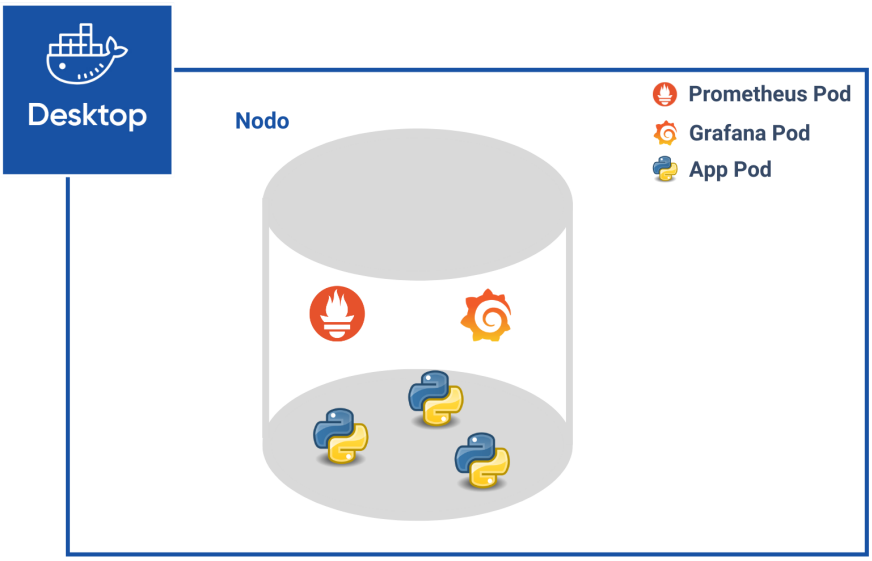

Figura 5.1: Docker-Desktop

5.2.1. Componentes del Despliegue

- **App.yml (Aplicación Flask)**: Este deployment gestiona la aplicación web Flask que sirve como la interfaz principal para los usuarios y la administración de entradas. La aplicación Flask es el corazón de nuestra plataforma, manejando todas las interacciones con los usuarios y procesando las transacciones de entradas.

- **Prometheus.yml (Prometheus)**: Prometheus es desplegado para monitorear el estado de la aplicación Flask y otros servicios dentro del clúster. Recopila métricas que son esenciales para el monitoreo de la salud del sistema y para asegurar que la performance de la aplicación se mantiene dentro de los parámetros óptimos.

- **Grafana.yml (Grafana)**: Grafana se utiliza para visualizar las métricas recogidas por Prometheus. Ofrece dashboards configurables que proporcionan insights visuales en tiempo real del rendimiento y la salud de la aplicación, facilitando la rápida detección y resolución de posibles problemas.

5.2.2. Simulación de Tráfico

Dado que los usuarios reales de la plataforma siguen interactuando con la plataforma monolítica existente y no tenemos acceso a datos de tráfico real de aquel entorno, es fundamental establecer una base de comportamiento de usuario en nuestro sistema en la nube. Esto es esencial para garantizar que, una vez se realice la migración total, la infraestructura en la nube pueda manejar eficientemente las interacciones de los usuarios sin problemas. Por lo

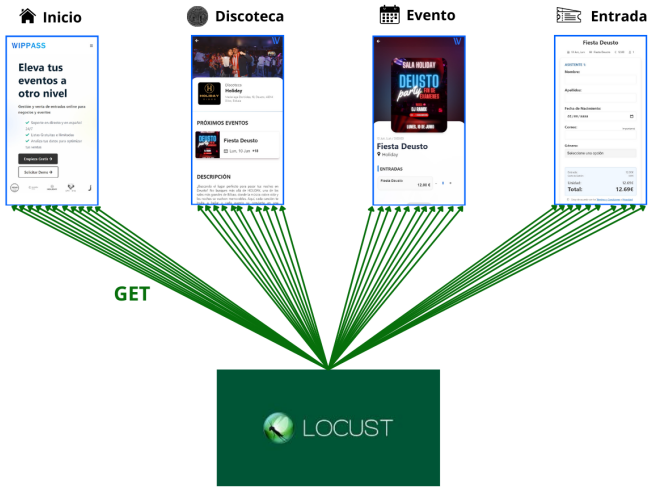

Figura 5.2: Visualización del tráfico generado por Locust, simulando el comportamiento habitual de los usuarios a través de las páginas más importantes, proporcionando una base de tráfico "sano"para futuras comparaciones.

tanto, la simulación de tráfico mediante Locust se convierte en un componente crucial del proceso de desarrollo y validación.

Locust se utiliza para generar un flujo continuo y controlado de solicitudes a las rutas más frecuentadas dentro de la aplicación, emulando el comportamiento normal de los usuarios en un día típico, sin picos de demanda ni errores. Las rutas son:

- Página de Inicio
- Página de organizador específico (como discotecas, promotoras, festivales, etc.)
- Página de evento específico
- Página de entrada específica

Esta simulación permite generar un patrón de tráfico constante y representativo que servirá como referencia o línea base para la detección futura de anomalías. Una vez que tengamos esta línea base, podremos introducir intencionalmente variaciones y errores en el sistema para ver cómo responde la infraestructura y cómo se detectan estas condiciones atípicas en comparación con el tráfico normal.

Adicionalmente, este enfoque nos permite evaluar la eficacia del Horizontal Pod Autoscaler (HPA) implementado en el deployment de nuestra aplicación. El HPA ajusta automáticamente

el número de pods en respuesta a las variaciones en la demanda detectadas durante la simulación, aumentando o disminuyendo los recursos asignados para mantener un rendimiento óptimo sin sobre-dimensionar la infraestructura.

Al establecer este modelo de tráfico "sano", nos aseguramos de que cualquier desviación del mismo pueda ser rápidamente identificada y analizada, facilitando la detección temprana de problemas potenciales que podrían surgir después de la migración completa de los usuarios a la nueva plataforma.

CAPÍTULO 6

Despliegue

6.1. Entorno de Producción (GKE)

Una de las principales ventajas de utilizar Kubernetes es su independencia respecto a la plataforma, lo que permite una gran flexibilidad en la elección del proveedor de servicios en la nube. Esta característica fue clave en la fase de planificación del despliegue en producción del proyecto. Los proveedores considerados incluyen:

- Rancher
- Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)
- Digital Ocean
- Red Hat

Aunque inicialmente se consideró utilizar Amazon Web Services (AWS) debido a su integración y servicios establecidos, se optó finalmente por Google Kubernetes Engine (GKE). La elección de GKE se basó en varios factores pragmáticos, destacando entre ellos el incentivo económico de Google Cloud Platform que ofrece un crédito de 300 dólares durante tres meses. Esta oferta resultó ser especialmente atractiva para el despliegue del proyecto, ya que proporciona los recursos necesarios sin coste inicial, permitiendo una implementación efectiva y una evaluación del rendimiento en un entorno real sin restricciones económicas inmediatas.

6.1.1. Ventajas de la Flexibilidad de Kubernetes

La capacidad de Kubernetes para operar a través de múltiples plataformas de nube es una de sus características más valiosas, permitiendo a los desarrolladores y a las empresas elegir la solución que mejor se adapte a sus necesidades específicas sin estar ligados a un proveedor específico. Esta flexibilidad no solo optimiza los costos, sino que también mejora la escalabilidad y la resilencia al permitir la portabilidad de las aplicaciones entre diferentes entornos de nube con mínima reconfiguración.

Esta estrategia asegura que el proyecto pueda beneficiarse de las últimas innovaciones en la gestión de contenedores y orquestación, al tiempo que se mantiene la agilidad para adaptarse a cambios futuros en tecnología o necesidades del proyecto.

6.2. Configuración del Entorno

Antes de desplegar cualquier recurso, es necesario preparar y configurar el entorno en Google Cloud Platform. Esto implica habilitar las API necesarias y configurar el entorno de trabajo para utilizar los servicios de Google Kubernetes Engine y Artifact Registry.

6.2.1. Entendiendo Google Cloud CLI y Cloud Shell

Google Cloud CLI es una herramienta de línea de comandos que permite a los usuarios gestionar recursos dentro de Google Cloud Platform (GCP) de manera eficiente. Facilita la ejecución de comandos para administrar proyectos y configuraciones sin tener que navegar a través de la interfaz gráfica de usuario.

Cloud Shell es una consola interactiva que se ejecuta en el navegador. Proporciona acceso a un entorno de línea de comandos preconfigurado en GCP, donde puedes ejecutar todos los comandos de **gcloud** y **kubectl**. Este entorno incluye herramientas de desarrollo preinstaladas y ofrece una capacidad temporal para alojar sesiones de desarrollo.

6.2.2. Configuración Inicial

Una vez habilitadas las API de Artifact Registry y Google Kubernetes Engine desde la consola de GCP, se puede proceder a configurar el proyecto y la zona utilizando Cloud Shell. Esta herramienta está disponible en la parte superior de la interfaz de la consola de GCP y se despliega en la parte inferior de la pantalla una vez activada.

- **Configuración del Proyecto:** Establece el proyecto actual en Google Cloud CLI con el siguiente comando:

```
$ gcloud config set project tfg-markel-420717

Updated property [core/project].
```

- **Definición de la Zona:** Especifica la zona de computación predeterminada para tus recursos, lo cual es crucial para la localización y gestión de instancias y servicios en GCP.

```
$ gcloud config set compute/zone us-central1-a

Updated property [compute/zone].
```

Al utilizar estos comandos, se establecen las configuraciones básicas necesarias para comenzar el despliegue y gestión de recursos en Google Kubernetes Engine, asegurando que todas las acciones se ejecuten dentro del contexto del proyecto y zona adecuados.

6.2.3. Crear el clúster

Un clúster consta de al menos una máquina de plano de control del clúster y varias máquinas trabajadoras llamadas nodos. Los nodos son instancias de máquinas virtuales (VM) de Compute Engine que ejecutan los procesos de Kubernetes necesarios para que sean parte del clúster. Las aplicaciones se despliegan en el clúster, y los pods que la constituyen se ejecutan en los nodos. [**?**]

```
$ gcloud container clusters create mi-cluster

Creating cluster mi-cluster in us-central1-a... Cluster is being
configured...working...
```

Después de crear el clúster, se deben obtener credenciales de autenticación para interactuar con él:

```
$ gcloud container clusters get-credentials mi-cluster

Fetching cluster endpoint and auth data.
kubeconfig entry generated for mi-cluster.
```

Hasta ahora, hemos creado un cluster de tres nodos, en la zona especificada. Se puede comprobar utilizando un comando especifico de **kubectl**:

```
$ kubectl get nodes

NAME                                          STATUS   ROLES    AGE     VERSION
gke-mi-cluster-default-pool-95adf183-87h7   Ready    <none>   3m12s   v1.28.8-gke.1095000
gke-mi-cluster-default-pool-95adf183-psz1   Ready    <none>   3m13s   v1.28.8-gke.1095000
gke-mi-cluster-default-pool-95adf183-rv25   Ready    <none>   3m13s   v1.28.8-gke.1095000
```

6.2.4. Preparación y Lanzamiento de Manifiestos

Una vez preparado el entorno, el siguiente paso es lanzar los manifiestos de Kubernetes que configuran y despliegan todos los recursos necesarios para la aplicación y los servicios de monitoreo. A continuación, se listan y describen los manifiestos utilizados:

- **app-deployment.yml**: Despliega la aplicación Flask en Kubernetes, gestionando las réplicas y asegurando su disponibilidad continua.

- **app-service.yml**: Define un servicio que expone la aplicación Flask dentro del clúster para facilitar el acceso interno y balanceo de carga.
- **hpa.yml**: Configura el Horizontal Pod Autoscaler para ajustar automáticamente el número de réplicas de la aplicación basado en la carga de trabajo actual.
- **prometheus-config.yml**: Contiene la configuración necesaria para que Prometheus monitoree adecuadamente los componentes del sistema.
- **prometheus-deployment.yml**: Despliega el servicio de Prometheus para recopilar métricas del clúster y las aplicaciones.
- **prometheus-service.yml**: Expone Prometheus dentro del clúster, facilitando la recolección de métricas.
- **grafana-config.yml**: Configura Grafana con los ajustes necesarios para integrarlo con Prometheus y visualizar las métricas recopiladas.
- **grafana-deployment.yml**: Despliega Grafana, que proporciona un dashboard para visualizar las métricas en tiempo real.
- **grafana-service.yml**: Define un servicio para exponer Grafana a los usuarios, permitiendo el acceso a los dashboards.
- **ingress.yml**: Configura las reglas de ingreso para dirigir el tráfico externo hacia los servicios adecuados dentro del clúster.
- **vigilante.yml**: Despliega un nuevo servicio encargado de la detección de anomalías, que evalúa las métricas recogidas y devuelve la puntuación de anomalía. Este componente será detallado más adelante.

Antes de proceder con el lanzamiento de los manifiestos de Kubernetes, es crucial asegurarse de que todas las imágenes de los contenedores necesarios estén disponibles en un repositorio accesible desde el clúster. Esto incluye la imagen de la aplicación Flask, que se debe alojar en un repositorio de imágenes como Google Container Registry (GCR) para garantizar su disponibilidad durante el despliegue.

6.2.5. Subida de Imágenes al Repositorio

Para subir la imagen de la aplicación Flask al GCR, primero se debe etiquetar la imagen local con el nombre del repositorio y luego subirla usando los siguientes comandos:

```
$ docker tag flask-app:latest gcr.io/tfg-markel-420717/flask-app:latest
$ docker push gcr.io/tfg-markel-420717/flask-app:latest
```

Una vez que la imagen está disponible en el repositorio, se puede proceder a actualizar el manifiesto del despliegue para que referencie correctamente la ubicación de la imagen:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask-app
        image: gcr.io/tfg-markel-420717/flask-app:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
        resources:
          requests:
            cpu: "250m"  # Solicita al menos 250 milicpu
```

Listing 6.1: Modificación del manifiesto app-deployment.yaml para utilizar la imagen desde GCR

Con la imagen ya disponible y el manifiesto actualizado, se pueden lanzar los manifiestos de Kubernetes. Esto se realiza a través de la siguiente serie de comandos ejecutados desde el Cloud Shell:

```
$ kubectl apply -f app-deployment.yml
$ kubectl apply -f app-service.yml
$ kubectl apply -f hpa.yml
$ kubectl apply -f prometheus-config.yml
$ kubectl apply -f prometheus-deployment.yml
$ kubectl apply -f prometheus-service.yml
$ kubectl apply -f grafana-config.yml
$ kubectl apply -f grafana-deployment.yml
$ kubectl apply -f grafana-service.yml
$ kubectl apply -f ingress.yml
$ kubectl apply -f vigilante.yml
```

Ahora, se listan todos los servicios, deployments y hpa que hay configurados:

```
$ kubectl get all
NAME                              READY   STATUS    RESTARTS   AGE
pod/flask-app-698ccd44cc-q84s4    1/1     Running   0          99s
pod/grafana-6dcb69fc6d-66kpq      0/1     Pending   0          19s
pod/prometheus-66cd8c85db-4q7hj   1/1     Running   0          52s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/flask-app    ClusterIP   10.92.65.30    <none>        5000/TCP   98s
service/grafana      ClusterIP   10.92.66.199   <none>        3000/TCP   26s
service/kubernetes   ClusterIP   10.92.64.1     <none>        443/TCP    15m
service/prometheus   ClusterIP   10.92.72.236   <none>        9090/TCP   38s
```

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flask-app     1/1     1            1           99s
deployment.apps/grafana       0/1     1            0           19s
deployment.apps/prometheus    1/1     1            1           52s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/flask-app-698ccd44cc       1         1         1       99s
replicaset.apps/grafana-6dcb69fc6d         1         1         0       19s
replicaset.apps/prometheus-66cd8c85db      1         1         1       52s

NAME                                              REFERENCE               TARGETS
MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/mi-hpa   Deployment/flask-app   0%/50%
1         10        1          86s
```

```
$ kubectl get ingress
NAME         CLASS   HOSTS   ADDRESS   PORTS   AGE
my-ingress   nginx   *                 80      2m54s
```

Se puede observar que no se le ha asignado una dirección IP al ingress, esto se debe a que no tenemos los controladores necesarios instalados.

```
$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
$ helm repo update
$ helm install nginx-ingress ingress-nginx/ingress-nginx --create-namespace
--namespace ingress-nginx
```

Después de instalar lo necesario y haber esperado lo necesario para que se apliquen los cambios, se puede observar como ya hay una IP asociada al ingress, la cual se va a utilizar para acceder a los diferentes servicios.

```
$ kubectl get ingress
NAME         CLASS   HOSTS   ADDRESS          PORTS   AGE
my-ingress   nginx   *       23.251.158.104   80      12m
```

6.2.6. Verificación del Funcionamiento de las Rutas con Ingress

Para asegurar que el enrutamiento a través del Ingress está configurado correctamente, se realizan pruebas específicas para cada servicio. El Ingress está diseñado para dirigir el tráfico hacia diferentes servicios basados en las rutas de acceso especificadas en su configuración.

6.2.6.1. Acceso al Servicio de Aplicación

Si se accede a la ruta `/login`, el Ingress debería dirigir el tráfico al servicio de la aplicación **app**. Se puede verificar esto con el siguiente comando:

```
$ curl 23.251.158.104/login

<!DOCTYPE html>
<html lang="es">
<head>
    <title>Login - Wippass</title> ...
```

Este comando debería mostrar la página de inicio de sesión de la aplicación, indicando que el tráfico se está redirigiendo correctamente.

6.2.6.2. Acceso al Servicio Grafana

De manera similar, al acceder a la ruta `/grafana`, el Ingress debería redirigir el tráfico al servicio **grafana**. Se puede probar con el siguiente comando:

```
$ curl 23.251.158.104/grafana

<html>
<head><title>503 Service Temporarily Unavailable</title></head>
```

En un escenario ideal, se espera que este comando devuelva la página de inicio de sesión de Grafana. Sin embargo, si el servicio no está disponible debido a recursos insuficientes en el cluster, podrías ver un error como el mostrado arriba. Esto indica que el tráfico se redirige correctamente, pero el pod de Grafana no puede iniciar debido a la insuficiencia de recursos del clúster.

6.2.7. Escalador Automático de Clúster

Para manejar de manera eficiente las fluctuaciones en la demanda de recursos del clúster, se implementa el **escalador automático de clúster [?]** en Google Kubernetes Engine (GKE). Este componente es esencial para mantener la alta disponibilidad y el rendimiento óptimo del clúster, ajustando dinámicamente la cantidad de nodos disponibles según las necesidades en tiempo real.

El escalador automático no ajusta los recursos de los nodos existentes (escalado vertical), sino que modifica el número total de nodos en el clúster (escalado horizontal). Funciona de la siguiente manera:

- **Expansión de nodos:** Cuando la carga de trabajo en el clúster aumenta y los recursos actuales son insuficientes para satisfacer la demanda, el escalador automático agrega más nodos al clúster. Esto asegura que todas las aplicaciones y servicios continúen operando sin interrupciones debido a la falta de recursos.
- **Reducción de nodos:** En períodos de baja demanda, para optimizar los costos y evitar el uso innecesario de recursos, el escalador automático puede retirar nodos, reduciendo el tamaño del clúster sin afectar el rendimiento de las aplicaciones que se ejecutan.

Esta capacidad de adaptación no solo mejora la eficiencia operativa sino también contribuye a una mejor gestión de costos, ya que solo se utilizan recursos cuando son estrictamente necesarios. A continuación, se muestra cómo habilitar y configurar el escalador automático para un clúster en GKE:

1. **Habilitar el ajuste de escala automático en GKE:** Este paso configura el escalado automático para el clúster, estableciendo el número de nodos entre un mínimo y un

máximo. El escalado automático es una característica soportada tanto por Kubernetes como por GKE, pero su implementación en GKE ofrece herramientas integradas que facilitan su configuración y manejo a través de la interfaz de línea de comandos de Google Cloud.

```
$ gcloud container clusters update mi-cluster --enable-autoscaling \
--min-nodes 3 --max-nodes 10

Updating mi-cluster...done.
```

2. **Optimizar la utilización de recursos mediante el perfil de escalado:** Tras habilitar el escalado automático, es recomendable ajustar el perfil de escalado a .ºptimize-utilization". Este perfil instruye al escalador automático para que priorice la eficiencia de los recursos sobre el mantenimiento de recursos adicionales inactivos. Esto ayuda a optimizar los costos al reducir el número de nodos en uso durante períodos de baja demanda, sin comprometer la capacidad de respuesta durante picos de carga.

```
$ gcloud container clusters update mi-cluster \
    --autoscaling-profile optimize-utilization

Updating mi-cluster...done.
```

Ahora, si se vuelven a imprimir los nodos disponibles, se puede observar como estos han aumentado:

```
$ kubectl get nodes

NAME                                          STATUS   ROLES    AGE     VERSION
gke-mi-cluster-default-pool-95adf183-87h7   Ready    <none>   156m    v1.28.8-gke.1095000
gke-mi-cluster-default-pool-95adf183-8gs2   Ready    <none>   74m     v1.28.8-gke.1095000
gke-mi-cluster-default-pool-95adf183-psz1   Ready    <none>   156m    v1.28.8-gke.1095000
gke-mi-cluster-default-pool-95adf183-rv25   Ready    <none>   156m    v1.28.8-gke.1095000
```

Solo faltaría eliminar el deployment de grafana y volver a lanzarlo:

```
$ kubectl delete -f grafana-deployment.yml

deployment.apps "grafana" deleted

$ kubectl apply -f grafana-deployment.yml

deployment.apps/grafana created
```

Finalmente, el servicio de grafana estará correctamente configurado:

```
$ curl 23.251.158.104/grafana

<a href="/grafana/login">Found</a>.
```

Cabe mencionar que grafana, al recibir una petición de un usuario que no ha iniciado sesión, lo redirige de /grafana a /login. En este entorno, eso es un problema, ya que el ingress al recibir la ruta /login, va a redirigir el tráfico a la pagina de login del servico de **app**. Hay que recordar que el ingress esta configurado para dirigir todo a **app** menos aquellas rutas que comienzen por /grafana.

Para resolver esto, es necesario configurar grafana, vía variables de entorno, para garantizar que siempre usa rutas que empiezan por /grafana".

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
      - name: grafana
        image: grafana/grafana:latest
        volumeMounts:
        - name: dashboard-volume
          mountPath: "/etc/grafana/provisioning/dashboards"
        ports:
        - containerPort: 3000
        resources:
          limits:
            memory: "2Gi"
            cpu: "1000m"
          requests:
            memory: "1Gi"
            cpu: "500m"
        env:
          - name: GF_SERVER_ROOT_URL
            value: "%(protocol)s://%(domain)s:%(http_port)s/grafana"
          - name: GF_SERVER_SERVE_FROM_SUB_PATH
            value: "true"
          - name: GF_SECURITY_ADMIN_USER
            value: "admin"
          - name: GF_SECURITY_ADMIN_PASSWORD
            value: "12345"
          - name: GF_DASHBOARDS_JSON_ENABLED
            value: "true"
          - name: GF_DASHBOARDS_JSON_PATH
            value: "/etc/grafana/provisioning/dashboards"
      volumes:
      - name: dashboard-volume
        configMap:
          name: grafana-dashboard
```

Listing 6.2: app-deployment.yaml

1. **GF_SERVER_ROOT_URL**: Esta variable establece la URL base de Grafana. Aquí, %(protocol)s:// %(domain)s: %(http_port)s/grafana asegura que Grafana espera ser servido desde un subpath (/grafana). Esta URL se usará para todas las redirecciones internas y la generación de URLs dentro de Grafana.

2. **GF_SERVER_SERVE_FROM_SUB_PATH**: Configurando esta variable a "true", le indicas a Grafana que será servido desde un subpath. Esto es crucial para asegurar que las redirecciones y enlaces internos de Grafana se generen correctamente.

6.2.8. Eliminar el Cluster

Eliminar el cluster para que no se apliquen cargos no deseados mientras no se estan haciendo pruebas:

```
$ gcloud container clusters delete mi-cluster

The following clusters will be deleted.
 - [mi-cluster] in [us-central1-a]

Do you want to continue (Y/n)?  Y

Deleting cluster mi-cluster...working...
```

CAPÍTULO 7

Observabilidad y Monitoreo

7.1. Monitoreo y Visualización

Este apartado explora las herramientas y técnicas empleadas para monitorear el comportamiento y la salud de la aplicación Flask desplegada en Kubernetes. La configuración adecuada de Prometheus y Grafana es vital para asegurar una visibilidad total del sistema y responder eficazmente a cualquier incidencia.

7.1.1. Configuración de Prometheus y Grafana

Prometheus se configura para recolectar métricas expuestas automáticamente por la aplicación Flask, utilizando la integración nativa de Prometheus. Las métricas clave, como **flask_http_request_duration_seconds**, son recolectadas de forma continua. Grafana se configura para consumir estas métricas desde Prometheus, utilizando **http://prometheus:9090** como fuente de datos principal. Esta integración asegura un flujo constante de datos y un monitoreo eficaz del sistema.

7.1.2. Métricas Clave

Prometheus recoge un conjunto de métricas específicas que son cruciales para evaluar el rendimiento de la aplicación. Estas incluyen:

- **flask_http_request_duration_seconds_bucket**: Histograma que mide la duración de las solicitudes HTTP.
- **flask_http_request_duration_seconds_count**: Contador total de todas las solicitudes HTTP recibidas.
- **flask_http_request_duration_seconds_created**: Timestamp de la creación de la métrica.

- **flask_http_request_duration_seconds_sum**: Suma total del tiempo de las solicitudes HTTP.
- **flask_http_request_total**: Contador total de solicitudes HTTP realizadas, incluyendo detalles por tipo de respuesta.

Estas métricas permiten monitorear desde la latencia y el tiempo total de las solicitudes hasta el conteo detallado de las mismas, ofreciendo una visión integral de la salud y rendimiento de la aplicación.

7.1.3. Detalles de los Gráficos de Grafana

El panel de visualización de Grafana se compone de varios gráficos que permiten monitorear diferentes aspectos del comportamiento y la salud de la aplicación Flask desplegada en Kubernetes. A continuación, se detallan los gráficos principales y las consultas PromQL utilizadas para generarlos:

- **Gráfico de Barras - Solicitudes Totales por Minuto**: Este gráfico utiliza la consulta Prometheus `sum(rate(flask_http_request_duration_seconds_count[1m]))` para calcular el número total de solicitudes por minuto. La métrica `flask_http_request_duration_seco` representa el conteo de todas las solicitudes HTTP recibidas, y el uso de `rate` sobre un intervalo de un minuto permite visualizar la tasa de solicitudes de forma agregada.
- **Gráfico de Línea - Tasa de Solicitudes Exitosas**: Este gráfico de línea muestra la tasa de solicitudes HTTP con estado 200, utilizando la consulta `rate(flask_http_request_duration_se` El gráfico ayuda a monitorizar la tasa de respuestas exitosas en intervalos de 30 segundos, ofreciendo una visión en tiempo real del éxito en la entrega de respuestas.
- **Gráfico de Área - Tasa de Errores**: Visualiza el total de errores, usando la consulta `sum(rate(flask_http_request_duration_seconds_count{status!="200"}[30s]))`. Este gráfico de área acumula las solicitudes que resultaron en cualquier estado distinto de 200, permitiendo identificar picos o incrementos inusuales en las respuestas erróneas.
- **Gráfico de Línea - Tiempo de Respuesta Medio por Endpoint**: Calcula el tiempo de respuesta medio para solicitudes exitosas, usando la fórmula `rate(flask_http_request_duration_ / rate(flask_http_request_duration_seconds_count{status="200"}[30s])`. Este gráfico ofrece una visión clara del rendimiento promedio de los endpoints, esencial para identificar degradaciones o mejoras en la respuesta de la aplicación.

Cada gráfico está meticulosamente diseñado para ser intuitivo y fácil de interpretar, lo que permite a los operadores hacer ajustes rápidos y fundamentados en los datos para optimizar la gestión y estabilidad del sistema.

Cada panel está diseñado para ser intuitivo y fácil de interpretar, permitiendo a los operadores realizar ajustes rápidos y basados en datos para optimizar la gestión del sistema.

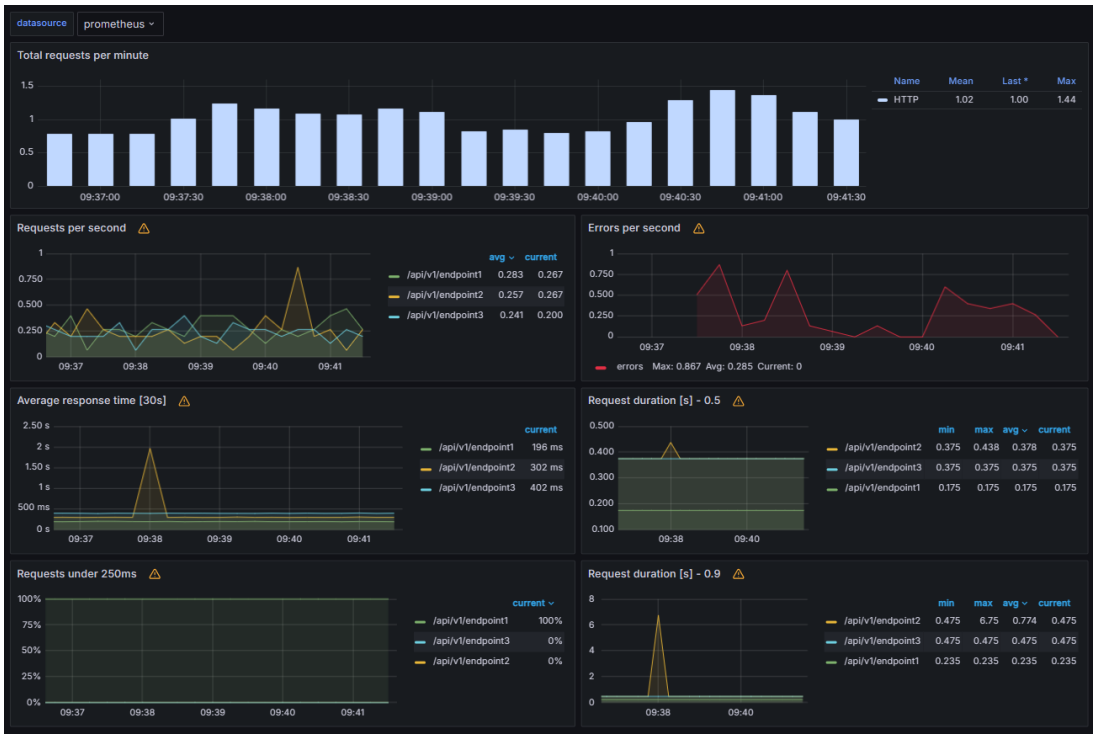

Figura 7.1: Panel predeterminado de Grafana para Flask mostrando métricas HTTP detalladas [1].

CAPÍTULO 8

Detección de Anomalias

En este proyecto, se definen como anomalías los picos de tráfico inusuales y el aumento considerable en el número de errores HTTP. Estas condiciones pueden indicar posibles problemas de rendimiento o ataques de denegación de servicio. La métrica principal utilizada para la detección es **flask_http_request_duration_seconds_count**, que mide la duración y la cantidad de solicitudes HTTP.

Utilizaremos dos consultas de PromQL para capturar los datos necesarios:

- **sum(increase(flask_http_request_duration_seconds_countstatus!='200'[1m])):** Cuenta el número de respuestas HTTP que resultaron en errores en el último minuto.
- **sum(increase(flask_http_request_duration_seconds_count[1m])):** Cuenta el total de peticiones HTTP en el último minuto.

8.1. Modelo de Detección de Anomalías: Half-Space Trees

El modelo seleccionado para la detección de anomalías en este proyecto es Half-Space Trees (HST), un enfoque robusto y eficiente especialmente adaptado para el análisis de flujos de datos en tiempo real. Este modelo es ideal para nuestro contexto debido a su capacidad de adaptarse rápidamente a cambios en los patrones de datos sin la necesidad de una reconstrucción completa del modelo [**?**].

8.1.1. Principios Operativos de Half-Space Trees

Los Half-Space Trees operan bajo un mecanismo de aprendizaje no supervisado que divide el espacio de datos en múltiples espacios o regiones utilizando una serie de árboles de decisión. Cada árbol en el modelo representa una partición del espacio de observaciones, y cada nodo dentro del árbol simboliza una subdivisión más específica basada en rangos de valores de datos (*vid.* Figura 8.1).

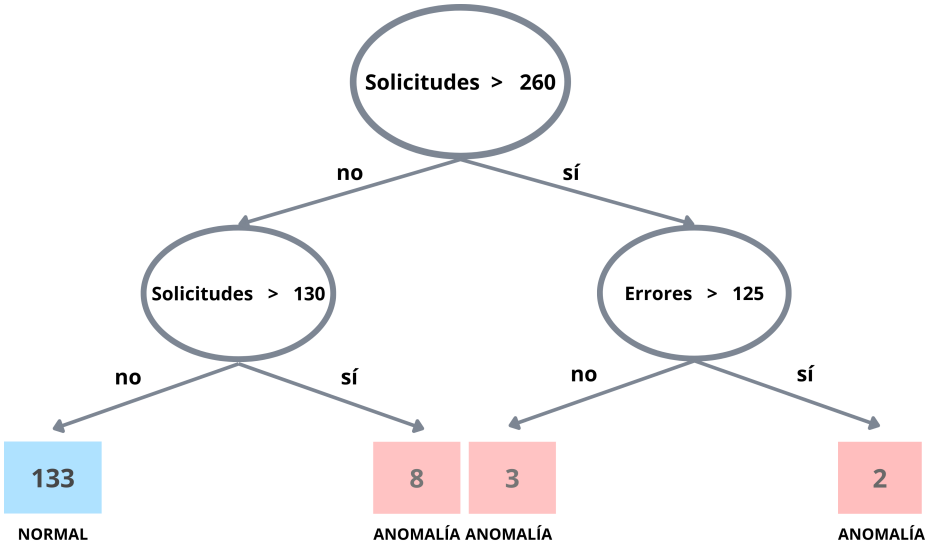

Figura 8.1: Visualización de cómo los Half-Space Trees particionan el espacio de datos, mostrando un ejemplo de las subdivisiones dentro de un árbol.

La construcción de los Half-Space Trees es notablemente eficiente. El modelo se construye dividiendo recursivamente el espacio de datos en hiperespacios más pequeños, permitiendo una rápida identificación de regiones con comportamientos atípicos. Esta eficiencia es crítica cuando se trabaja con grandes volúmenes de datos en tiempo real, donde la velocidad de procesamiento es crucial.

8.1.2. Análisis en Tiempo Real y Detección de Anomalías

Cada nuevo punto de datos se evalúa en tiempo real a medida que atraviesa el modelo de árboles. Al llegar a un nodo específico, el punto se compara con los perfiles históricos de los datos de ese nodo. Si el punto se desvía significativamente de los patrones históricos, se considera una anomalía. Esta capacidad de análisis instantáneo es esencial para sistemas que requieren respuestas rápidas ante condiciones cambiantes (*vid.* Figura 8.2).

8.2. Implementación y Entrenamiento Inicial del Modelo

El proceso de entrenamiento inicial del modelo Half-Space Trees es crucial para establecer una base de conocimiento sobre lo que constituye un tráfico "normal". Este entrenamiento se lleva a cabo mediante una simulación de tráfico constante durante una hora, lo que permite al modelo aprender y adaptarse sin generar falsas alarmas una vez que esté operativo.

8.2.1. Recolección de Datos para el Entrenamiento Inicial

Durante la fase de entrenamiento inicial, se extraen métricas de la aplicación Flask cada 15 segundos. Estas métricas, que incluyen el total de peticiones HTTP y los errores HTTP, son

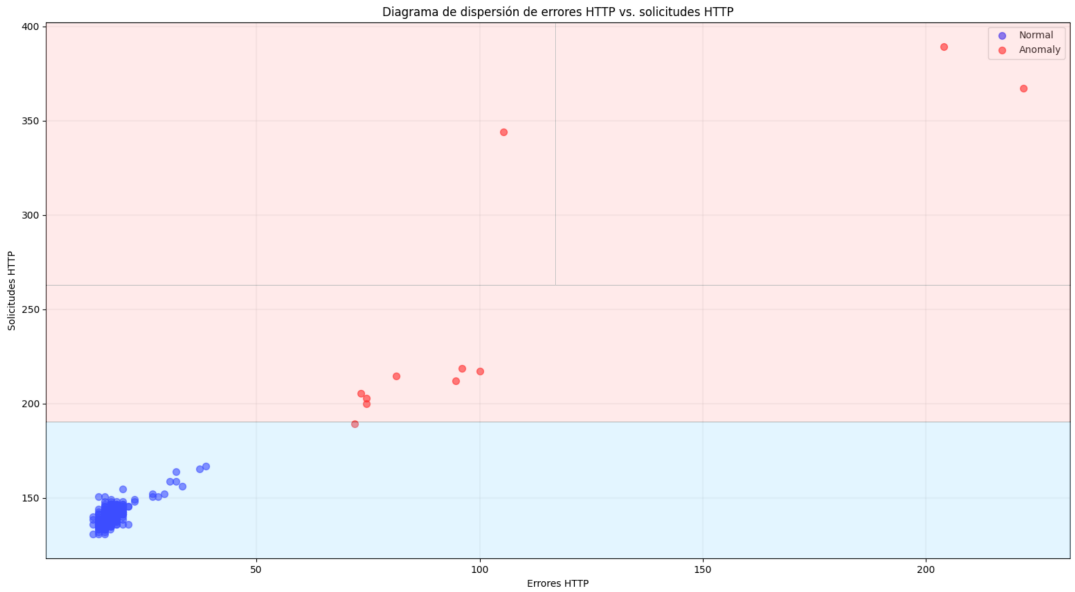

Figura 8.2: Ejemplo de diagrama de dispersión mostrando cómo los datos son analizados por los Half-Space Trees. La área azul representa la region con mayor frecuencia de puntos, mientras que las áreas rojas podrían indicar anomalías.

http_errors	***http_total_requests***
18.66	140.00
16.00	138.66
14.66	130.66

Tabla 8.1: Tres primeras instancias almacenadas que representan el tráfico típico sin anomalías. Estos datos son fundamentales para configurar el modelo antes de su despliegue real.

esenciales para que el modelo comprenda los patrones de tráfico normales bajo condiciones controladas. Los datos se almacenan en un archivo tipo *csv*, que luego se utiliza para entrenar el modelo.

Es importante destacar que este entrenamiento inicial es solo el comienzo del ciclo de aprendizaje del modelo. Una vez desplegado, el modelo sigue refinando su capacidad para detectar anomalías mediante el aprendizaje continuo de nuevas métricas recolectadas durante su operación normal. Este aprendizaje en curso permite al modelo adaptarse a cambios en los patrones de tráfico y mantener su precisión a lo largo del tiempo.

Este enfoque proactivo y preparatorio asegura que el modelo de detección de anomalías esté bien equipado para identificar desviaciones significativas sin ser perturbado por fluctuaciones normales en el tráfico de datos. Así, se mejora significativamente la confiabilidad y efectividad del sistema de monitoreo de anomalías desde su implementación inicial.

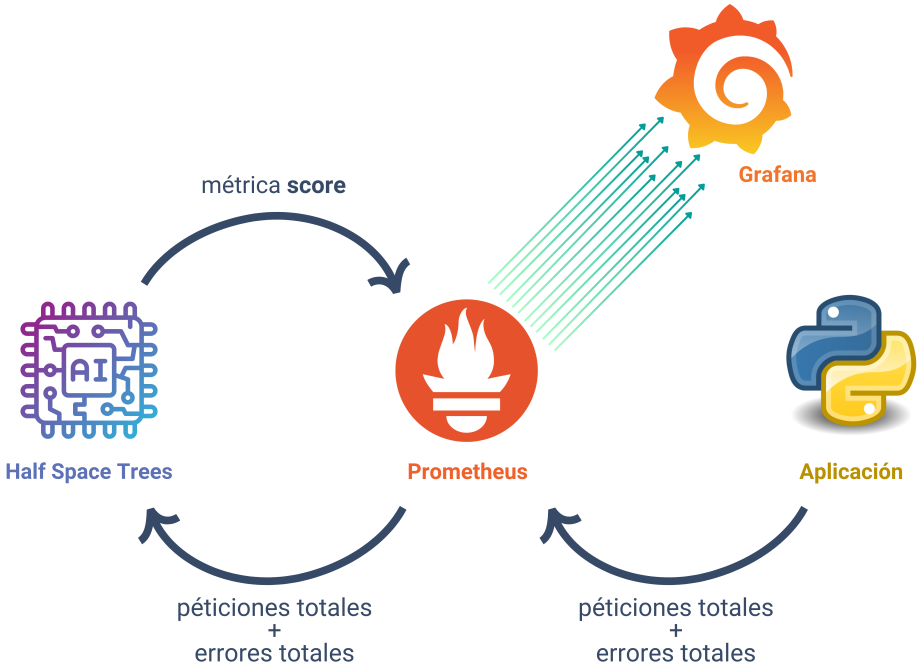

Figura 8.3: Esquema de la integración del servicio de detección de anomalías en Kubernetes, mostrando la recolección y procesamiento de métricas.

8.3. Integración en Kubernetes y Continuo Monitoreo

Para integrar el modelo de detección de anomalías en el entorno de producción, se ha creado un nuevo servicio dentro de Kubernetes. Este servicio, denominado "vigilante", opera de forma continua, evaluando las métricas recogidas y calculando una puntuación de anomalía que refleja el estado actual del tráfico.

El servicio vigilante es implementado como un deployment en Kubernetes, configurado para interactuar con Prometheus. Se establece una métrica *score*, la cual representa el nivel de anomalía detectado y puede aumentar o disminuir, dependiendo del tráfico observado.

Para capturar las métricas relevantes, se añaden reglas de scrapeo en Prometheus. Estas reglas están configuradas no solo para recolectar métricas estándar de Flask, sino también para extraer la puntuación de anomalía del servicio vigilante. Esta configuración asegura que todas las métricas necesarias para el análisis estén disponibles en Prometheus.

En Grafana, se configura un nuevo panel para visualizar el nivel de anomalía en tiempo real. Este panel ayuda a los operadores a monitorizar el estado del sistema y a identificar rápidamente cualquier comportamiento inusual que pueda requerir intervención (*vid.* Figura 8.4).

Esta integración de Half-Space Trees, Kubernetes, Prometheus y Grafana crea un robusto sistema de detección y monitoreo de anomalías, optimizado para responder de manera eficaz a las dinámicas del tráfico web en tiempo real.

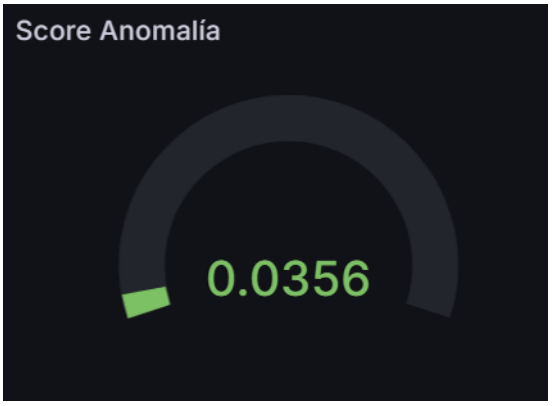

Figura 8.4: Gráfico que indica el nivel de anomalía detectado en cada momento.

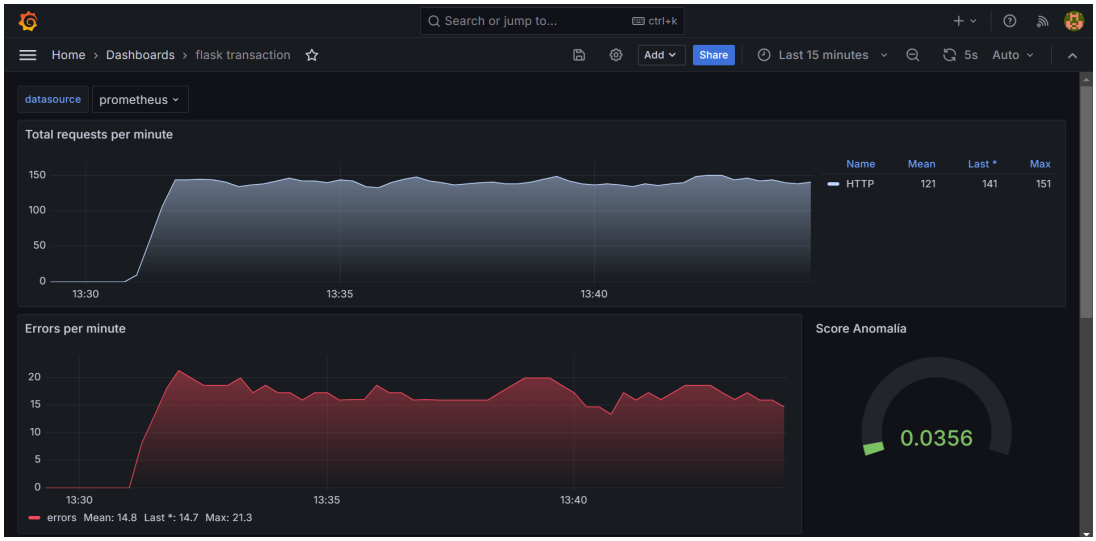

Figura 8.5: Visualización del tráfico constante y sin anomalías. Se muestran tanto las peticiones totales como los errores HTTP totales a lo largo de una hora, con una puntuación de anomalía cercana a cero, indicando un comportamiento típico y esperado.

8.4. Resultados y Análisis

La implementación del modelo Half-Space Trees y su integración en el entorno de monitoreo han proporcionado resultados significativos en la detección de anomalías. Los gráficos subsecuentes ilustran dos situaciones distintas: tráfico normal y un evento de anomalía.

La Figura 8.5 muestra el comportamiento del tráfico bajo condiciones normales, donde no se detectan anomalías. Este gráfico es crucial porque establece una línea base del tráfico esperado en el sistema. Observamos que las peticiones totales y los errores HTTP se mantienen dentro de un rango predefinido, lo cual indica un funcionamiento estable de la aplicación.

Contrastando con el tráfico normal, la Figura 8.6 muestra un pico inesperado tanto en las

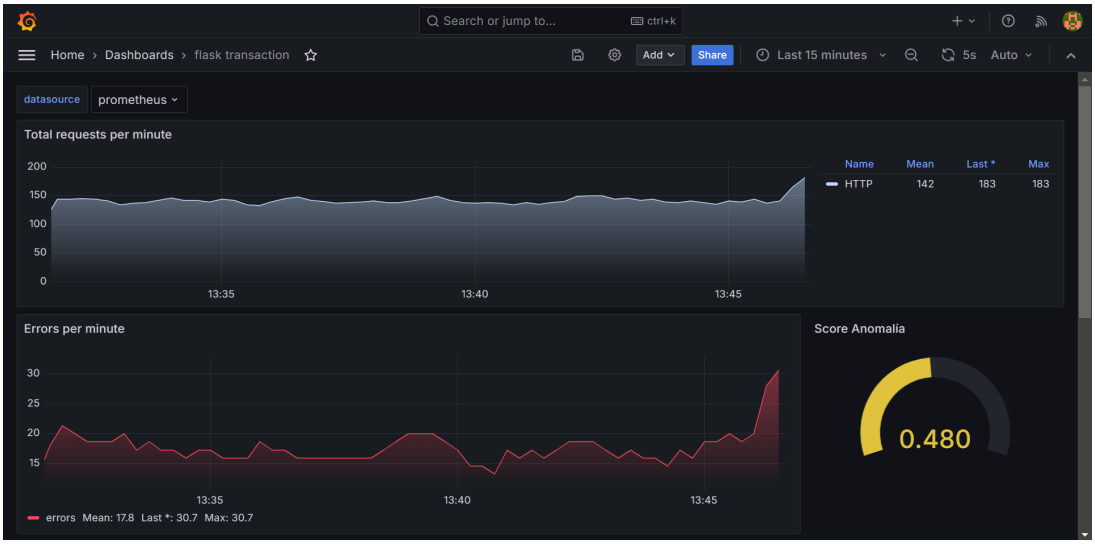

Figura 8.6: Visualización de picos de tráfico. Este gráfico muestra variaciones significativas respecto al tráfico habitual, con un aumento notable en la puntuación de anomalía a 0.48, indicando una posible intervención necesaria.

peticiones totales como en los errores HTTP. Este pico es identificado por el modelo como una anomalía, reflejado por una puntuación de 0.48, lo cual es considerablemente más alto que lo usual. Este evento sugiere un potencial problema técnico o un ataque de denegación de servicio, demostrando la eficacia del modelo en identificar y alertar sobre cambios significativos en el patrón de tráfico.

8.5. Posibles Mejoras

A medida que avanzamos en el desarrollo y refinamiento de nuestro sistema de detección de anomalías, identificamos varias áreas para posibles mejoras:

La incorporación de más métricas podría enriquecer la capacidad del modelo para discernir entre anomalías verdaderas y fluctuaciones normales del tráfico. Métricas adicionales podrían incluir tiempos de respuesta del servidor, tasas de éxito de las solicitudes y análisis más detallados del comportamiento del usuario.

La utilización de múltiples modelos de detección en un enfoque conjunto podría mejorar la precisión y reducir los falsos positivos. Cada modelo podría especializarse en diferentes aspectos del tráfico o del comportamiento de la aplicación, proporcionando una visión más holística y robusta.

El ajuste continuo del modelo en respuesta a nuevos datos recopilados permitirá al sistema adaptarse mejor a las condiciones cambiantes del entorno de la aplicación. Este proceso de aprendizaje continuo es vital para mantener la relevancia y efectividad del modelo frente a la evolución de las pautas de tráfico y los cambios en la infraestructura de la aplicación.

Estas mejoras no solo aumentarán la eficacia del sistema actual, sino que también prepararán la infraestructura para responder de manera más eficiente a los desafíos futuros, asegurando que la plataforma siga siendo segura y confiable para los usuarios.

CAPÍTULO 9

Resultados y Verificación

CAPÍTULO 10

Conclusiones y Trabajo Futuro

10.1. Conclusiones

El proyecto de reingeniería descrito en este trabajo ha permitido aumentar significativamente la disponibilidad de la aplicación Flask, mejorando su escalabilidad y la capacidad de gestionar eficientemente los picos de demanda mediante el uso de Kubernetes. La integración de herramientas de observabilidad avanzadas como Prometheus y Grafana ha proporcionado una visión detallada del rendimiento de la aplicación en tiempo real, lo que ha facilitado la rápida identificación y resolución de problemas.

Sin embargo, a pesar de estos avances, la aplicación sigue siendo monolítica. Esto puede limitar su capacidad para adaptarse rápidamente a cambios y escalar de manera eficiente conforme a las necesidades específicas de sus diversos componentes.

10.2. Trabajo Futuro

Para superar las limitaciones de la arquitectura monolítica y optimizar aún más la flexibilidad y escalabilidad de la aplicación, el próximo paso será migrar hacia una arquitectura de microservicios. Este enfoque permitirá una mejor separación de preocupaciones, facilitando el mantenimiento y la actualización independiente de cada componente funcional de la aplicación.

10.2.1. Plan de Migración a Microservicios

- **Identificación de Componentes:** El primer paso será identificar los diferentes módulos o componentes de la aplicación actual. Esto incluye la autenticación (Auth), el sistema de gestión de contenidos (Blog), la interfaz de usuario (Frontend), la lógica de negocio (Backend) y las APIs que conectan el frontend con el backend.

- **Desacoplamiento de Servicios:** Cada uno de estos componentes será desarrollado y desplegado como un servicio independiente, lo que permitirá escalarlos de acuerdo a sus necesidades específicas sin afectar a los demás componentes.

- **Integración Continua y Despliegue Continuo (CI/CD):** Se implementarán prácticas de CI/CD para automatizar las pruebas y el despliegue de cada microservicio, asegurando que las actualizaciones puedan ser implementadas de manera rápida y fiable.

La migración hacia una arquitectura de microservicios es fundamental para superar las limitaciones actuales de la aplicación monolítica y para optimizar la escalabilidad y flexibilidad del sistema. Sin embargo, existen otros aspectos críticos que también deben abordarse para asegurar el rendimiento y la escalabilidad de la plataforma.

10.2.2. Desacoplamiento de la Base de Datos

Aunque hemos logrado aumentar la disponibilidad de la aplicación Flask desplegándola en múltiples pods dentro de Kubernetes, todos estos pods actualmente acceden a una única base de datos compartida. Este diseño presenta un cuello de botella significativo en términos de escalabilidad y resiliencia:

- **Replicación y Particionamiento de la Base de Datos:** Para mitigar este riesgo y mejorar el rendimiento, es esencial considerar estrategias como la replicación y particionamiento de la base de datos. Esto no solo ayudará a distribuir la carga sino también a mejorar la disponibilidad y la tolerancia a fallos del sistema de almacenamiento de datos.

- **Bases de Datos como Servicio:** Explorar opciones de bases de datos como servicio (DBaaS) que puedan escalar automáticamente y gestionar de manera más eficiente el acceso concurrente a los datos por múltiples instancias de la aplicación.

10.2.3. Monitorización Avanzada con Grafana

Para complementar la reestructuración arquitectónica, también es crucial mejorar nuestras capacidades de monitorización:

- **Métricas de Kubernetes en Grafana:** Integrar métricas específicas de Kubernetes, como contadores de pods y nodos, directamente en Grafana. Esto permitirá visualizar y analizar el estado y la salud de la infraestructura de Kubernetes sin depender exclusivamente de comandos de terminal.

- **Alertas Proactivas:** Configurar alertas proactivas basadas en estas métricas para identificar y responder automáticamente a problemas como saturación de recursos o fallos en nodos.

10.2.4. Conclusión y Perspectivas Futuras

La transición a microservicios y la mejora en la gestión de la base de datos y la monitorización son pasos cruciales para que nuestra plataforma pueda adaptarse y crecer de manera sostenible. Estas mejoras no solo aumentarán la eficiencia operativa y la capacidad de respuesta de la aplicación, sino que también fortalecerán nuestra infraestructura para enfrentar desafíos futuros y aprovechar nuevas oportunidades.

Al abordar tanto los aspectos arquitectónicos como los operativos, nuestro proyecto estará mejor posicionado para mantener un rendimiento óptimo y una alta disponibilidad en un entorno de mercado dinámico y competitivo.

Estas mejoras no solo aumentarán la escalabilidad y la eficiencia operativa de la aplicación, sino que también mejorarán la capacidad de la plataforma para innovar y adaptarse a las cambiantes demandas del mercado.

10.2.5. Retos y Consideraciones

- **Complejidad de la Gestión:** La migración a microservicios conlleva desafíos en la gestión de la infraestructura, especialmente en lo que respecta a la orquestación de servicios, la seguridad y la comunicación entre servicios.
- **Monitorización y Observabilidad:** Será crucial desarrollar estrategias robustas de monitorización y observabilidad para manejar la complejidad aumentada y asegurar la alta disponibilidad y el rendimiento de la plataforma.

En conclusión, aunque la transición a microservicios presenta retos significativos, los beneficios potenciales en términos de escalabilidad, mantenibilidad y agilidad justifican la inversión y el esfuerzo requeridos. A medida que la aplicación evoluciona hacia esta arquitectura más moderna, esperamos no solo mejorar la eficiencia operativa sino también la experiencia del usuario y la capacidad de respuesta a las necesidades del negocio.

Implementando esta estrategia, la plataforma no solo estará mejor equipada para manejar las exigencias de tráfico variable y el crecimiento continuo, sino que también podrá aprovechar las ventajas de la modularidad para innovar y expandirse de manera más efectiva en el futuro.

Apéndice

Eranskinak

Bibliografía

[1] Grafana contributors. Dashboard de transacciones para flask en grafana, 2024. Accessed: 2024-06-06.

[2] Wikipedia contributors. Flask, 2024. Accessed: 2024-05-28.

[3] Locust contributors. Locust - a modern load testing framework, 2024. Accessed: 2024-03-03.

[4] Kubernetes Authors. Kubernetes documentation, 2023. Accessed: 2024-05-28.

[5] Ihor Dvoretskyi Nuno do Carmo. Wsl+docker: Kubernetes on the windows desktop, 2020. Accessed: 2024-02-01.

[6] Google Cloud Authors. Gke documentation, 2024. Accessed: 2024-04-12.

[7] Google Cloud Skills Boost Authors. Comprende y combina las estrategias de ajuste de escala automático de gke, 2024. Accessed: 2024-04-12.

[8] S. C. Tan, K. M. Ting, and T. F. Liu. Fast anomaly detection for streaming data. *IJCAI Proceedings — International Joint Conference on Artificial Intelligence*, 22(1):1511–1516, 2011.