**Progreso 2 – Integración de Sistemas**

**Caso Plataforma de Servicios Estudiantiles**

Martín Vargas

**Repositorio Github:**

La universidad desea construir una nueva plataforma integrada para atender solicitudes académicas de los estudiantes (solicitudes de certificados, legalizaciones, homologaciones y equivalencias). Actualmente, existen 3 sistemas independientes:

1. Sistema Académico (REST API – gestionado internamente)

2. Sistema de Certificación (SOAP – externo, expuesto por un proveedor estatal)

3. Sistema de Seguridad y Roles (interno, usa tokens JWT)

Tu tarea es diseñar e implementar una solución de integración funcional utilizando los

conocimientos adquiridos durante el curso.

**Objetivos específicos**

1. Integrar servicios REST y SOAP en una solución funcional.

2. Exponer todos los servicios a través de un API Gateway.

3. Diseñar la solución considerando aspectos de trazabilidad, seguridad y resiliencia.

4. Aplicar patrones como Circuit Breaking y Retry usando conceptos de Service Mesh (en

forma de diseño o pseudocódigo si no se puede desplegar, con una evaluación menor

que si se lo implementa).

**Diagrama Alto Nivel:**

**Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.**

La arquitectura propuesta para la Plataforma de Servicios Estudiantiles se centra en un **API Gateway (Kong)** como punto único de entrada. Este gateway gestionará la seguridad inicial (autenticación mediante API Key, rate limiting), la trazabilidad básica y el enrutamiento de las solicitudes hacia los microservicios internos.

El microservicio principal a desarrollar es **SolicitudService (REST API)**, implementado con Python/Flask. Este servicio:

1. Recibirá las solicitudes de los estudiantes (previamente autenticadas por el API Gateway).
2. Validará internamente el token JWT proporcionado en la cabecera Authorization de la solicitud. Esta validación se realiza usando un secreto compartido y verifica la firma y expiración del token. Para este examen, esta lógica de validación de JWT reside dentro del SolicitudService.
3. Interactuará con el **Sistema de Certificación (SOAP externo)** para registrar la certificación. Esta interacción es simulada en el código e incluye una capa de resiliencia mediante un patrón de **Retry** (implementado con la librería Tenacity) para manejar fallos temporales del servicio externo.
4. Potencialmente, podría interactuar con el **Sistema Académico (REST API interno)** si fuera necesario para obtener datos adicionales del estudiante, aunque esta interacción no se implementa en detalle para el alcance de este examen.

**Puntos Clave de la Implementación:**

* **Seguridad:**
  + **API Gateway (Kong):** Se configura con un plugin key-auth. Los clientes deben presentar una API Key válida en la cabecera apikey para acceder a los servicios enrutados a través de Kong. Adicionalmente, se aplica una política de rate-limiting para proteger los servicios backend de un exceso de solicitudes.
  + **SolicitudService:** Implementa la validación de tokens JWT. Espera un token JWT en la cabecera Authorization: Bearer <token>. Este token es validado (firma, expiración) usando un secreto compartido (JWT\_SECRET). Esta es la capa de autorización a nivel de aplicación, asegurando que solo usuarios con un JWT válido puedan realizar operaciones.
* **Trazabilidad:**
  + **API Gateway:** Kong puede ser configurado (aunque no detallado explícitamente en el kong.yaml de este examen para simplificar) para inyectar un ID de correlación (Trace ID) en las cabeceras HTTP que se propagan a los servicios downstream.
  + **SolicitudService:** Los logs generados por la aplicación Flask (visibles en la salida del contenedor Docker) incluyen información que permitiría seguir el flujo de una solicitud (ej. IDs de solicitud, mensajes de INFO/ERROR). En un sistema más avanzado, se integrarían librerías de trazabilidad distribuida como OpenTelemetry.
* **Resiliencia (Circuit Breaking y Retry):**
  + **SolicitudService:** Se ha implementado una política de reintentos (Retry) usando la librería tenacity de Python para la llamada simulada al Sistema de Certificación SOAP. Esto ayuda a manejar fallos transitorios del servicio externo. El patrón de Circuit Breaking se discute conceptualmente para un entorno más complejo (como Istio en Kubernetes), ya que su implementación completa en un setup Docker-Compose simple excede el alcance.

Este diseño busca un equilibrio entre la seguridad en el borde (gestionada por el API Gateway) y la seguridad y lógica de negocio a nivel de aplicación (gestionada por SolicitudService), complementado con mecanismos básicos de resiliencia para las interacciones con servicios externos.

Capturas de Codigo:

Estructura del proyecto:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Kong.yaml:  
Texto

El contenido generado por IA puede ser incorrecto.

Main.py:

Texto

El contenido generado por IA puede ser incorrecto.

Security.py:  
Texto

El contenido generado por IA puede ser incorrecto.

Dockerfile:  
Texto

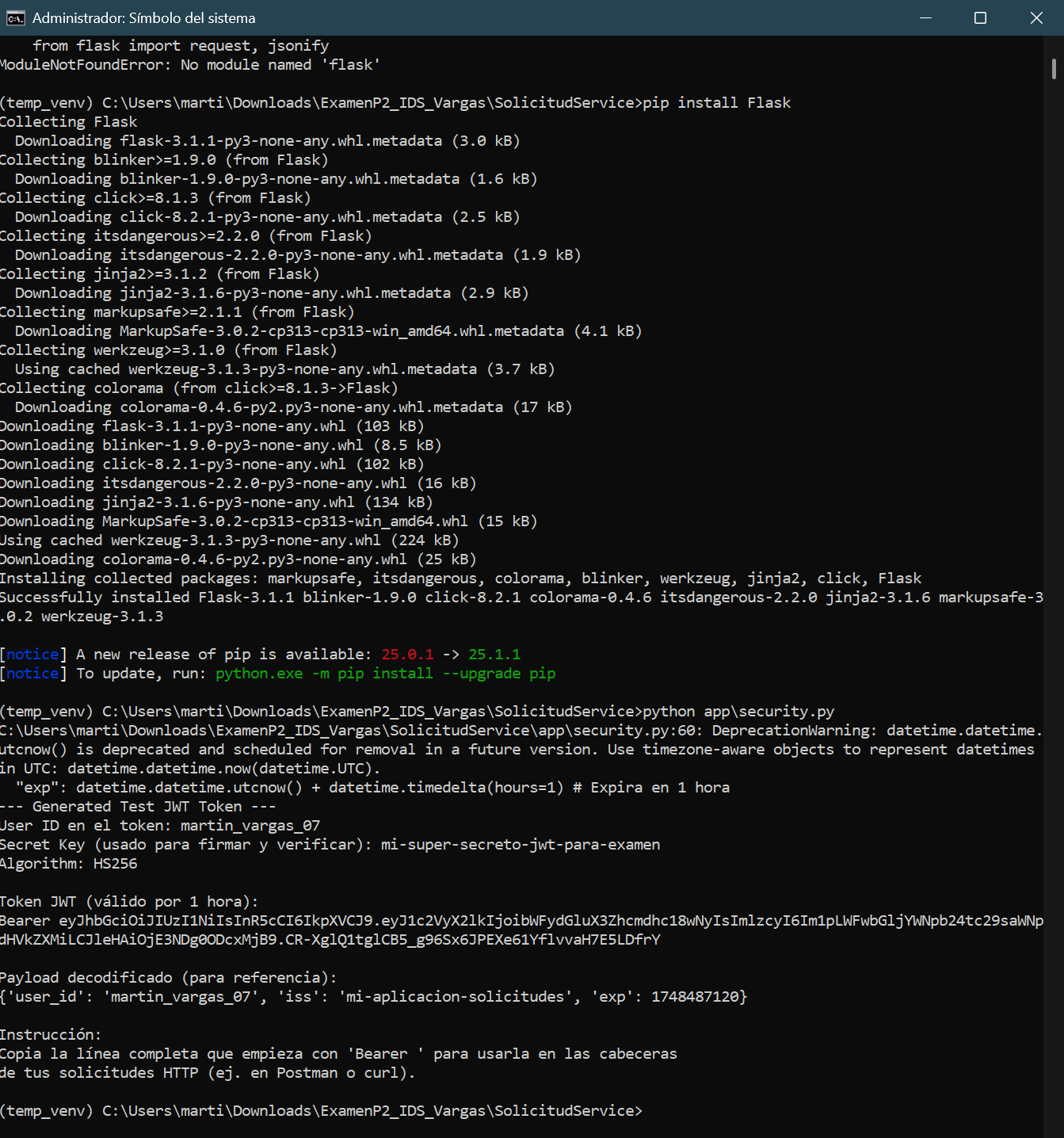
El contenido generado por IA puede ser incorrecto.  
  
Dockercompose:

Texto

El contenido generado por IA puede ser incorrecto.

**Implementación:**

Generacion del Token:



Levantamos Docker compose:

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Se verifica que estén Se verifica que estén arriba los servicios(también existen de proyectos anteriores):

Texto

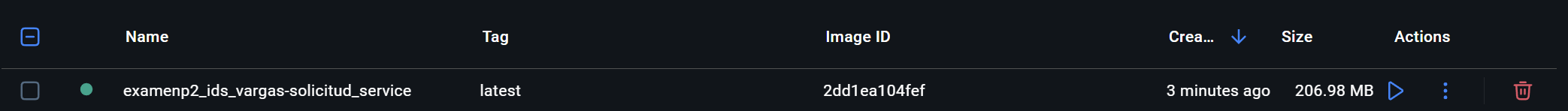
El contenido generado por IA puede ser incorrecto.

Contenedores en Docker:

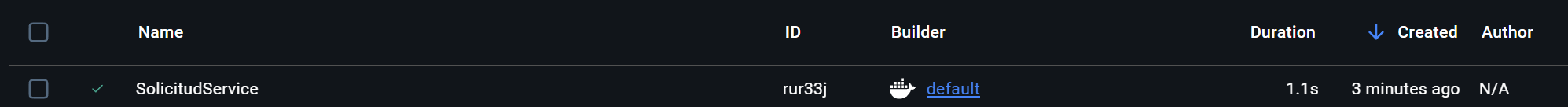
Interfaz de usuario gráfica, Texto, Aplicación

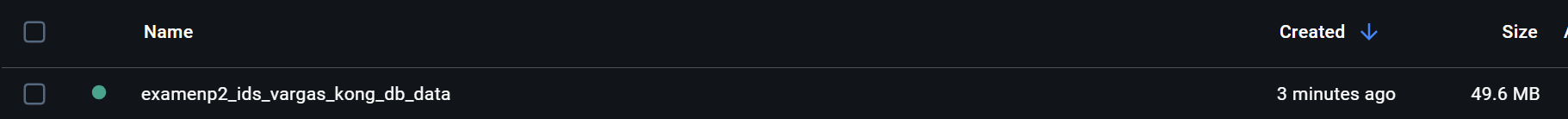
El contenido generado por IA puede ser incorrecto.

Imagen:



Build y volumen:





Creamos costumer y api key:

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Flask funcionando:

Texto

El contenido generado por IA puede ser incorrecto.

Se intento varias veces y se cambio varias veces el Kong.yaml:

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Por mas que estén up los contenedores:

**Implementación de Circuit Breaking y Retry**

**Retry Automático al Servicio SOAP (Implementado en SolicitudService)**

Se ha implementado una política de reintentos directamente en el SolicitudService al llamar al sistema SOAP externo (simulado). Se utiliza la librería tenacity de Python, que proporciona un decorador @retry fácil de usar para añadir esta funcionalidad.

Configuración Aplicada en el Código:

* **Máximo de Intentos:** La función que simula la llamada SOAP (llamar\_sistema\_soap\_externo\_con\_retry) se intentará un máximo de **2 veces** en total. Esto significa un intento original y, si este falla, un reintento adicional. Esto se configura con el parámetro stop=stop\_after\_attempt(2) en el decorador @retry.
* **Espera entre Intentos:** Si un intento falla, el sistema esperará **1 segundo** fijo antes de realizar el siguiente intento. Esto se configura con wait=wait\_fixed(1).
* **Condición para Reintentar:** Los reintentos solo se activarán si la función decorada lanza una excepción específica, que en este caso hemos definido como SoapCallFailedError. Esto se configura con retry=retry\_if\_exception\_type(SoapCallFailedError).
* **Re-lanzar la Excepción (Reraise):** Si todos los intentos (el original y los reintentos) fallan, la excepción original (SoapCallFailedError) es finalmente re-lanzada. Esto es importante para que el código que llama a la función pueda capturar el error y manejarlo apropiadamente (por ejemplo, para registrar el error detalladamente o devolver un estado de error específico al cliente de la API). Esto se asegura con el parámetro reraise=True.

Fragmento de código relevante de SolicitudService/app/main.py:

*from tenacity import retry, stop\_after\_attempt, wait\_fixed, retry\_if\_exception\_type*

*class SoapCallFailedError(Exception):*

*pass*

*@retry(*

*stop=stop\_after\_attempt(2),*

*wait=wait\_fixed(1),*

*retry=retry\_if\_exception\_type(SoapCallFailedError),*

*reraise=True # Importante para que el error se propague si los reintentos fallan*

*)*

*def llamar\_sistema\_soap\_externo\_con\_retry(solicitud\_data):*

*print(f"INFO: Intentando llamada a sistema SOAP para datos: {solicitud\_data}")*

*# ... simulación de llamada ...*

*if random.random() < 0.5: # Simula fallo el 50% de las veces*

*print("ERROR: Llamada SOAP simulada: FALLO")*

*raise SoapCallFailedError("El servicio SOAP simulado no respondió correctamente o está temporalmente inaccesible.")*

*else:*

*print("INFO: Llamada* SOAP simulada: ÉXITO")

return True

**Circuit Breaker (Diseño Conceptual con Istio)**

El patrón Circuit Breaker es más complejo de implementar directamente en una aplicación simple sin librerías adicionales o un service mesh. Si esta solución estuviera desplegada en un entorno de Kubernetes gestionado por **Istio Service Mesh** (lo cual está fuera del alcance de esta implementación basada en Docker Compose), la configuración del Circuit Breaker se realizaría de forma declarativa a través de recursos personalizados de Istio, específicamente un DestinationRule.

Objetivo del Requerimiento: Abrir el circuito si hay más de 3 fallos al servicio SOAP en 60 segundos.

Ejemplo de DestinationRule (Pseudocódigo YAML para Istio): Supongamos que el servicio SOAP externo es accesible dentro de la malla de servicios de Istio bajo el nombre de host soap-certification-service.external.svc.cluster.local.

*apiVersion: networking.istio.io/v1beta1*

*kind: DestinationRule*

*metadata:*

*name: soap-certification-service-circuit-breaker*

*# namespace: el-namespace-de-tus-servicios # Especificar el namespace donde aplica*

*spec:*

*host: soap-certification-service.external.svc.cluster.local # El FQDN del servicio externo*

*trafficPolicy:*

*connectionPool: # Configuraciones opcionales pero recomendadas para la gestión de conexiones*

*tcp:*

*maxConnections: 100 # Límite de conexiones TCP simultáneas*

*http:*

*http1MaxPendingRequests: 100 # Límite de solicitudes HTTP/1.1 pendientes*

*maxRequestsPerConnection: 10 # Límite de solicitudes por conexión*

*outlierDetection: # Aquí se define la configuración del Circuit Breaker*

*# Si Istio observa 3 errores 5xx consecutivos al llamar a una instancia de este servicio,*

*# esa instancia se considera "no saludable" y se expulsa temporalmente del pool de balanceo de carga.*

*consecutive5xxErrors: 3*

*# Intervalo de tiempo para los análisis de detección de outliers. Istio revisa la salud*

*# de las instancias en estos intervalos.*

*interval: "10s" # Ejemplo: cada 10 segundos*

*# Duración mínima por la cual una instancia es expulsada del pool de balanceo de carga*

*# una vez que se detecta como "no saludable". Durante este tiempo, el circuito está "abierto"*

*# para esa instancia.*

*baseEjectionTime: "60s" # Ejemplo: expulsado por 60 segundos*

*# Porcentaje máximo de instancias en el pool de balanceo de carga que pueden ser expulsadas.*

*# Si se establece en 100, todas las instancias pueden ser expulsadas si todas fallan.*

*maxEjectionPercent: 100*

*# minHealthPercent: 0 # Opcional: umbral mínimo de hosts saludables para mantener el servicio operativo.*

Explicación del YAML de Istio para el Circuit Breaker:

* host: Especifica el servicio upstream al que se aplicará esta política de tráfico.
* outlierDetection: Contiene la configuración específica del Circuit Breaker.
  + consecutive5xxErrors: 3: El proxy de Istio (Envoy) abrirá el circuito para una instancia del servicio si detecta 3 errores HTTP de la serie 5xx (ej. 500, 502, 503, 504) consecutivos.
  + interval: "10s": Istio realiza los análisis de detección de outliers (instancias problemáticas) en intervalos de 10 segundos. Es durante estos intervalos que se cuentan los consecutive5xxErrors.
  + baseEjectionTime: "60s": Una vez que una instancia es expulsada (el circuito para esa instancia se "abre"), permanecerá fuera del pool de balanceo de carga durante al menos 60 segundos. Después de este tiempo, Istio puede permitir que una pequeña cantidad de tráfico pase a la instancia expulsada para verificar si se ha recuperado (estado "semi-abierto").
  + maxEjectionPercent: 100: Permite que Istio expulse hasta el 100% de las instancias del servicio si todas están fallando consistentemente.

Interpretación para el requisito "más de 3 fallos en 60 segundos": La configuración de Istio con consecutive5xxErrors: 3 y baseEjectionTime: "60s" aborda este requisito. Si ocurren 3 fallos consecutivos (detectados en un intervalo de hasta 10s, según interval), la instancia se expulsa inmediatamente por 60 segundos. Si los fallos no son estrictamente consecutivos pero ocurren múltiples fallos que llevan a consecutive5xxErrors dentro de los interval de chequeo, se activará la expulsión. La clave es que una vez que se alcanza el umbral de fallos consecutivos, el circuito se abre por el baseEjectionTime especificado, protegiendo al servicio cliente de seguir intentando contra un servicio defectuoso.

**Métricas y Trazas Clave a Capturar**

Métricas Clave:

* Desde **Kong API Gateway**:
  + Latencia de las solicitudes (tiempo total de procesamiento en el gateway, y latencia añadida por Kong).
  + Rendimiento (Throughput): Número de solicitudes por segundo/minuto.
  + Uso de plugins: Métricas de plugins específicos como rate-limiting (número de solicitudes limitadas) o key-auth (intentos de autenticación fallidos/exitosos).
  + Estado de los upstreams (servicios backend como SolicitudService): Salud (up/down) y disponibilidad.
* Desde **SolicitudService**:
  + Latencia de endpoints: Tiempo de respuesta promedio y percentiles (p95, p99) para POST /solicitudes y GET /solicitudes/{id}.
  + Tasa de errores por endpoint (errores 4xx y 5xx generados por la aplicación).
  + Rendimiento por endpoint (solicitudes por segundo).
  + Estado del Circuit Breaker (si se implementara a nivel de aplicación): Número de veces que el circuito se abre, se cierra o está en estado semi-abierto; número de llamadas bloqueadas por el circuito.
  + Conteos de reintentos: Número de reintentos realizados por tenacity al servicio SOAP.
  + Uso de recursos de la aplicación/contenedor: Uso de CPU, uso de memoria, descriptores de archivo abiertos, etc. (métricas a nivel de JVM si fuera Java, o del intérprete Python).

Trazas (Seguimiento de una solicitud individual a través del sistema):

* **Flujo completo de la solicitud:** Se registraría cada "salto" o componente involucrado en el procesamiento de una solicitud:
  1. Llegada de la solicitud del usuario al API Gateway (Kong).
  2. Procesamiento en Kong (autenticación con API Key, rate limiting, enrutamiento).
  3. Reenvío de Kong a SolicitudService.
  4. Procesamiento en SolicitudService (validación de JWT, lógica de negocio).
  5. Llamada de SolicitudService al Sistema de Certificación SOAP (simulado), incluyendo los reintentos si ocurren.
  6. Propagación de las respuestas de vuelta a través de la misma cadena hasta el usuario.
* **Identificación de cuellos de botella:** Analizando los "spans" (que representan una unidad de trabajo o una operación, como una llamada HTTP o una consulta a base de datos) y sus duraciones dentro de una traza, se puede identificar qué componente o llamada está tardando más y contribuyendo a la latencia general.
* **Análisis de errores:** Cuando una solicitud falla, la traza distribuida puede ayudar a identificar en qué punto de la cadena de llamadas ocurrió el error y con qué contexto (parámetros de la solicitud, errores de servicios dependientes, etc.), facilitando la depuración.

**Implementación conceptualmente:**

1. **Instrumentación del Código:**
   * SolicitudService: Se añadiría la librería prometheus\_flask\_exporter y se configuraría para exponer un endpoint /metrics. Para la trazabilidad, se integrarían las librerías del SDK de OpenTelemetry y la instrumentación automática para Flask (opentelemetry-instrumentation-flask). Se configuraría un exportador de OTel para enviar las trazas a Jaeger o Zipkin.
   * Kong: Se habilitaría el plugin de Prometheus. Para la trazabilidad, se configuraría el plugin de Zipkin, Jaeger, o DataDog (que a menudo soportan la propagación de contextos de traza OpenTelemetry, o se usaría un plugin específico de OTel para Kong si estuviera disponible y maduro).
2. **Despliegue de la Infraestructura de Observabilidad:**
   * Se desplegarían los componentes de la pila de observabilidad (Prometheus, Grafana, Jaeger/Zipkin, y ELK/Loki si se elige esa pila) preferiblemente como contenedores Docker. Estos podrían ser gestionados por el mismo archivo docker-compose.yml para un entorno de desarrollo/examen, o en un clúster Kubernetes para un entorno de producción.
3. **Configuración de las Herramientas:**
   * Se configuraría Prometheus para que haga "scraping" (recolección periódica) de métricas desde los endpoints /metrics expuestos por Kong y SolicitudService.
   * Se configuraría Grafana añadiendo Prometheus como "datasource" y luego creando dashboards para visualizar las métricas deseadas (ej. dashboards para el rendimiento de Kong, salud de SolicitudService, etc.).
   * Se configurarían los SDKs de OpenTelemetry en SolicitudService para que exporten las trazas generadas al colector o agente del sistema de trazabilidad elegido (Jaeger/Zipkin).
   * Se configurarían los agentes de logging (o la configuración de Docker para los drivers de logging de los contenedores) para enviar los logs de los contenedores a la pila de logging centralizado.