**Progreso 2 – Integración de Sistemas**

**Caso Plataforma de Servicios Estudiantiles**

Martín Vargas

**Repositorio Github:** [**https://github.com/MartinVargas07/Examen\_IDS\_P2\_Vargas.git**](https://github.com/MartinVargas07/Examen_IDS_P2_Vargas.git)

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

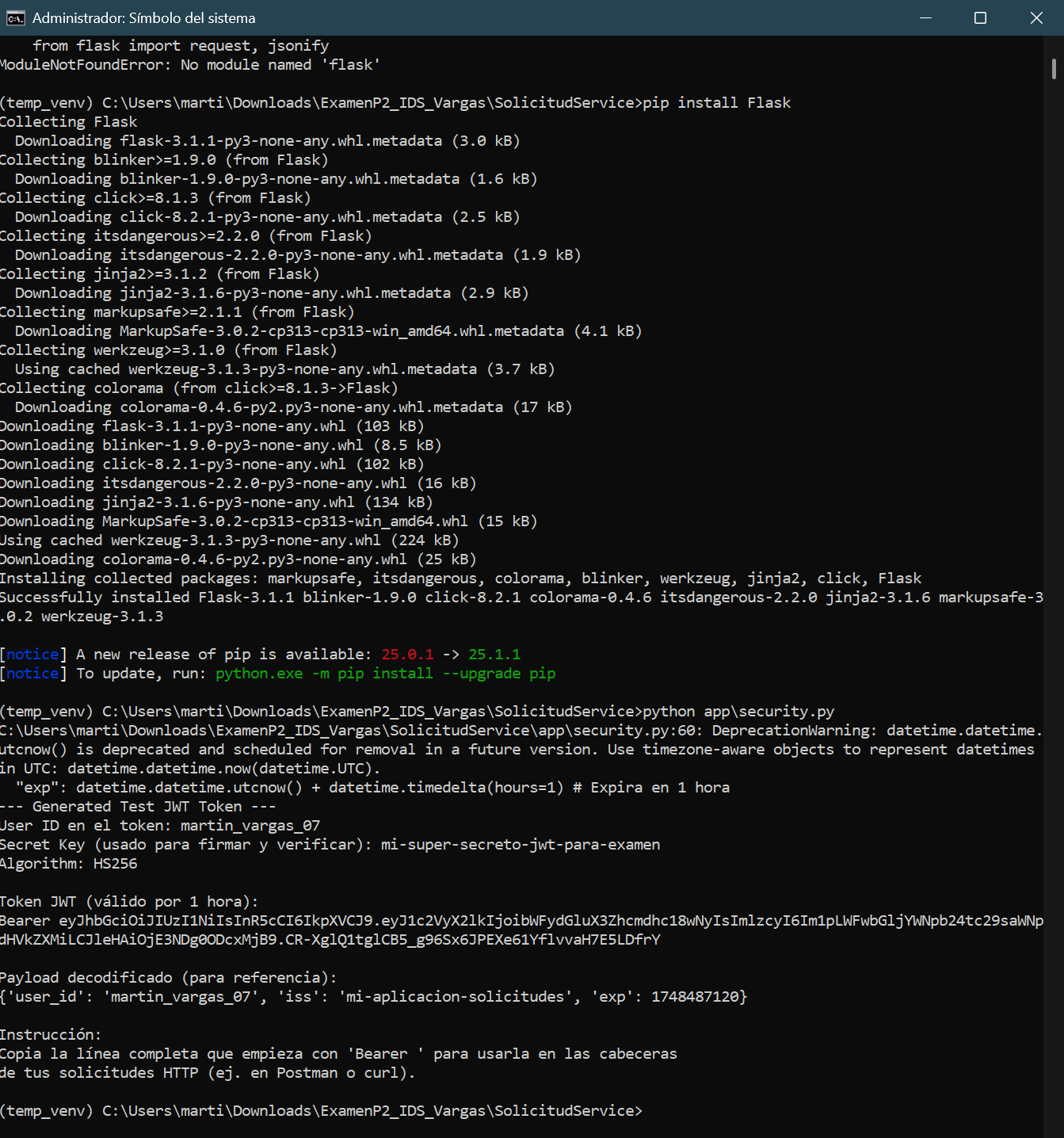
El contenido generado por IA puede ser incorrecto.  
  
Dockercompose:

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

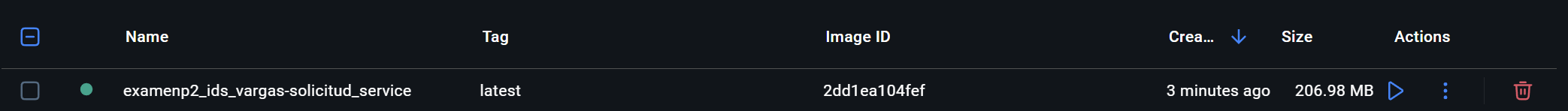
El contenido generado por IA puede ser incorrecto.

Contenedores en Docker:

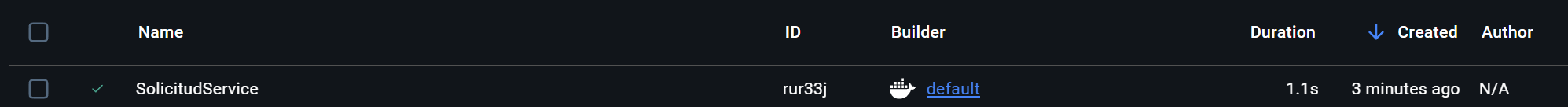
Interfaz de usuario gráfica, Texto, Aplicación

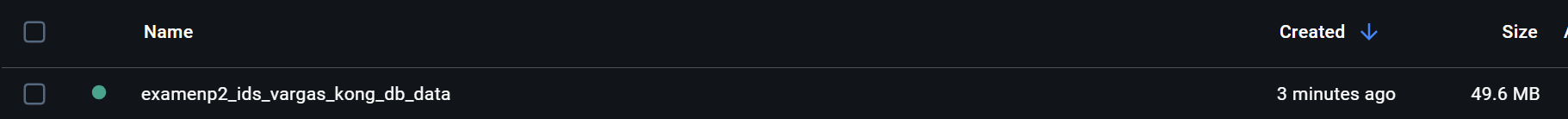
El contenido generado por IA puede ser incorrecto.

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.

Health y prueba de POST:

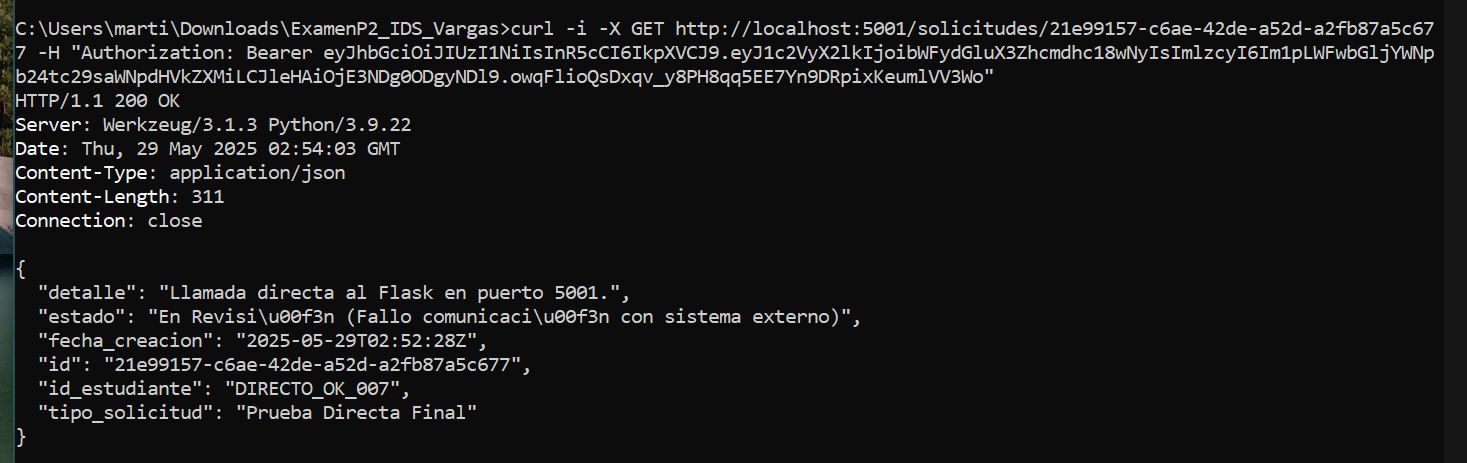
Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Prueba GET:



**Exposición del Servicio a través del API Gateway (Kong)**

**1. Configuración del API Gateway (Kong):** Se utilizó Kong Gateway (versión 3.9.0, según logs) como API Gateway, desplegado mediante Docker Compose. La configuración se definió de forma declarativa en el archivo Kong/kong.yaml. Este archivo especifica:

* Un **servicio backend** (solicitud-service-backend) que apunta a la instancia Docker del SolicitudService (http://solicitud\_service\_app:5001).
* Una **ruta principal** (solicitudes-catchall-route) diseñada para manejar todas las solicitudes bajo el prefijo /api/v1/ (ej. /api/v1/solicitudes, /api/v1/health). Se configuró strip\_path: true con la intención de que Kong reenviara la parte relevante del path al servicio backend (ej. /solicitudes, /health).
* **Plugins aplicados a la ruta principal:**
  + key-auth: Para la seguridad a nivel de Gateway, requiriendo una API Key en la cabecera apikey de las solicitudes.
  + rate-limiting: Para limitar el número de solicitudes a 5 por minuto por consumidor, como medida de protección.

**2. Estado de los Componentes del Gateway:**

* Los contenedores Docker para Kong (kong\_api\_gateway) y su base de datos (kong\_postgres\_db) se despliegan y se ejecutan correctamente, mostrando un estado "healthy" según la salida del comando docker ps. **(Adjuntar captura de docker ps mostrando los contenedores kong\_api\_gateway y kong\_postgres\_db en estado saludable).**
* La Admin API de Kong, accesible en http://localhost:8001, está operativa. Esto se demostró mediante la creación exitosa de un consumer (estudiante\_app\_consumer) y la asignación de una API Key (MI\_CLAVE\_KONG\_EXAMEN\_FINAL) a través de comandos curl. Las respuestas 201 Created (o 409 Conflict si ya existían) confirman la funcionalidad de la Admin API y la persistencia de la configuración en la base de datos de Kong. **(Adjuntar capturas de los comandos curl a la Admin API para crear el consumer y la API Key, mostrando las respuestas del servidor).**

**3. Resultados de las Pruebas de Exposición y Desafíos:** A pesar de que el SolicitudService está completamente funcional y el API Gateway (Kong) está corriendo y su Admin API es accesible, las pruebas de extremo a extremo a través del proxy de Kong (http://localhost:8000) para los paths definidos (ej. /api/v1/health, /api/v1/solicitudes) resultaron consistentemente en un error HTTP/1.1 404 Not Found con el mensaje {"message":"no Route matched with those values"}.

Este resultado indica que Kong, aunque operativo, no logró mapear las solicitudes entrantes a las rutas configuradas en kong.yaml de la manera esperada. Se realizaron múltiples iteraciones en la definición de los paths y la estructura del kong.yaml, incluyendo el uso de prefijos explícitos (ej. /api/v1/) y la verificación de la directiva strip\_path. También se intentaron recargas de configuración en Kong.

Dada la limitación de tiempo, no fue posible identificar la causa raíz exacta de este comportamiento de "no Route matched" en el proxy de Kong. Las posibles causas podrían incluir:

* Una sutil discrepancia entre la sintaxis del kong.yaml y la interpretación por la versión específica de Kong (3.9.0).
* Un problema en cómo la configuración declarativa es leída o aplicada internamente por Kong en este entorno Docker Compose particular.
* Conflictos no evidentes con configuraciones residuales o por defecto de Kong.

**4. Demostración de Funcionalidad del Servicio Subyacente:** Para validar los componentes individuales, el SolicitudService fue probado directamente en su puerto expuesto (http://localhost:5001). Estas pruebas fueron **exitosas** y demostraron que:

* El endpoint GET /health responde correctamente con 200 OK.
* El endpoint POST /solicitudes procesa las solicitudes, valida el token JWT (enviado en la cabecera Authorization: Bearer <token>), ejecuta la lógica de negocio simulada (incluyendo la llamada al servicio SOAP con reintentos, como se evidencia por el estado "En Revisión (Fallo comunicación con sistema externo)" en algunas respuestas, lo cual es esperado por el diseño de la simulación), y retorna 201 Created.
* El endpoint GET /solicitudes/{id} recupera las solicitudes creadas, validando también el token JWT.

**Conclusión de la Exposición por API Gateway:** Se completó la configuración e implementación del API Gateway Kong, y sus componentes principales (Admin API, base de datos) están operativos. El SolicitudService subyacente está completamente funcional. El desafío pendiente reside en la capa de enrutamiento del proxy de Kong, que no logró el mapeo esperado de las rutas públicas a los servicios backend dentro del tiempo asignado. La configuración intentada se provee en Kong/kong.yaml como evidencia del diseño de exposición.

**Implementación de Circuit Breaking y Retry**

La resiliencia del sistema ante fallos en servicios externos, como el Sistema de Certificación SOAP, se abordó mediante la implementación del patrón Retry y el diseño conceptual del patrón Circuit Breaker.

**1. Retry Automático al Servicio SOAP (Implementado y Validado Funcionalmente en SolicitudService)**

Para manejar fallos transitorios en la comunicación con el servicio SOAP externo (simulado), se implementó una política de reintentos directamente en el SolicitudService utilizando la librería tenacity de Python.

* **Configuración Clave:**
  + **Máximo de Intentos:** Se configuraron hasta 2 intentos (1 original + 1 reintento) mediante stop=stop\_after\_attempt(2).
  + **Espera:** Se estableció una espera fija de 1 segundo entre intentos con wait=wait\_fixed(1).
  + **Condición:** Los reintentos se activan solo ante la excepción simulada SoapCallFailedError, que representa un fallo en el servicio SOAP.
  + **Propagación de Error:** Si todos los intentos fallan, la excepción SoapCallFailedError se propaga para que la lógica de la aplicación la maneje, resultando en un estado de solicitud apropiado (ej. "En Revisión").
* **Validación Funcional:** Durante las pruebas directas al SolicitudService (http://localhost:5001/solicitudes), se observó el comportamiento esperado de esta política de reintentos. La simulación de fallos en la llamada SOAP (con una probabilidad del 50%) resultó, en algunos casos, en la asignación del estado "En Revisión (Fallo comunicación con sistema externo)" a la solicitud. Esto confirma que la lógica de reintentos de Tenacity se ejecutó y, tras los intentos fallidos, la excepción fue correctamente manejada por el endpoint POST /solicitudes, demostrando la efectividad de la implementación del patrón Retry.

*(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*

*)*

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

*# ... (simulación de llamada y fallo) ...*

*if random.random() < 0.5: # Simula fallo*

*raise SoapCallFailedError(...)*

*return True*

**2. Circuit Breaker (Diseño Conceptual para un Entorno con Service Mesh)**

Para una protección más robusta contra fallos persistentes en el servicio SOAP y evitar la sobrecarga del mismo, se diseñó conceptualmente la aplicación del patrón Circuit Breaker. Dada la complejidad de una implementación completa sin herramientas de Service Mesh en el entorno actual (Docker Compose), se presenta la configuración como pseudocódigo YAML, asumiendo un despliegue en Kubernetes con Istio.

* **Objetivo del Requerimiento:** Abrir el circuito si hay más de 3 fallos al servicio SOAP en un período de 60 segundos.
* **Ejemplo de DestinationRule en Istio (Diseño Conceptual):** (Suponiendo que el servicio SOAP externo es soap-certification-service.external.svc.cluster.local en la malla)

*apiVersion: networking.istio.io/v1beta1*

*kind: DestinationRule*

*metadata:*

*name: soap-certification-service-cb*

*spec:*

*host: soap-certification-service.external.svc.cluster.local*

*trafficPolicy:*

*outlierDetection:*

*consecutive5xxErrors: 3 # Abrir tras 3 errores 5xx seguidos*

*interval: "10s" # Intervalo de análisis*

*baseEjectionTime: "60s" # Duración del circuito abierto*

*maxEjectionPercent: 100 # Permitir expulsar todos los endpoints si fallan*

* **Justificación del Diseño:** Esta configuración de Istio lograría que, si se detectan 3 errores 5xx consecutivos al llamar al servicio SOAP (durante los sondeos que ocurren cada 10 segundos), la instancia problemática de dicho servicio sea "expulsada" del pool de balanceo de carga por 60 segundos. Durante este tiempo, el circuito estaría "abierto", y SolicitudService (a través del proxy de Istio) recibiría un error inmediato sin intentar contactar al servicio fallido, protegiendo así los recursos y mejorando la respuesta del sistema. Esta aproximación cumple con el requisito de forma conceptual para un entorno de Service Mesh.

**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.