# Pruebas de Software Servicio OpenAI SmishGuard

# 1. Introducción

**Propósito**: Documentar las pruebas realizadas para validar el correcto funcionamiento del servicio de API basado en Flask y OpenAI. Este documento describe pruebas unitarias, de carga y de API para los endpoints `/`, `/ping`, `/consultar-modelo-gpt` y `/conclusion-modelo-gpt`.

**Alcance de las pruebas**:

- **Pruebas Unitarias**: Validación de la funcionalidad individual de los endpoints, incluyendo el manejo de errores.
- **Pruebas de Carga**: Evaluación del rendimiento bajo diferentes cargas de usuario.
- **Pruebas de API**: Validación de la API en cuanto a respuestas esperadas, manejo de errores y tiempos de respuesta.

**Objetivos**:

- Asegurar la estabilidad de los endpoints.
- Validar el rendimiento bajo carga.
- Asegurar que el servicio maneja errores y excepciones correctamente.

# 2. Metodología de Pruebas

**Tipos de pruebas**:

1. **Unitarias**: Validar la funcionalidad de los endpoints individuales.
2. **Carga**: Verificar el rendimiento del sistema cuando es utilizado por múltiples usuarios simultáneamente.
3. **API**: Asegurar la correcta respuesta de los endpoints, incluyendo validación de datos y manejo de excepciones.

**Herramientas de prueba**:

- **Pruebas Unitarias**: `unittest` para pruebas de unidad en Python.
- **Pruebas de Carga**: `Locust` para simular múltiples usuarios.
- **Pruebas de API**: `pytest` para pruebas funcionales de los endpoints.

**Criterios de aceptación**:

- **Unitarias**: Cada endpoint debe devolver los códigos de estado y mensajes esperados.
- **Carga**: El sistema debe soportar cargas de hasta 100 usuarios concurrentes con tiempos de respuesta aceptables.

- **API**: Responder con los datos esperados y manejar adecuadamente los casos de errores de entrada.

# 3. Pruebas Unitarias

**Propósito**: Validar la funcionalidad de cada endpoint en la API.

**Herramientas**: `unittest` y `pytest`.

## Estrategia de Pruebas

### Cobertura de Pruebas:

- Cobertura de código de al menos el 80%.

### Criterios de Ejecución:

- Ejecutar pruebas mediante `unittest` y `pytest` en un entorno de desarrollo.

## Casos de Prueba

### 3.1 `/` Endpoint

- **Identificador**: `test_hello_world`
- **Descripción**: Validar que el endpoint `/` responde correctamente.
- **Entradas**: Ninguna.
- **Acciones**: Realizar solicitud `GET` a `/`.
- **Resultados Esperados**: Código de estado `200` y respuesta `"Hello, world!"`.

### 3.2 `/ping` Endpoint

- **Identificador**: `test_ping`
- **Descripción**: Validar la respuesta del endpoint `/ping`.
- **Entradas**: Ninguna.
- **Acciones**: Realizar solicitud `GET` a `/ping`.
- **Resultados Esperados**: Código de estado `200` y mensaje `{"message": "pong"}`.

### 3.3 `/consultar-modelo-gpt` Endpoint

- **Identificador**: `test_consultar_modelo`
- **Descripción**: Validar la respuesta cuando se envía un mensaje.
- **Entradas**: `{ "mensaje": "¿Es seguro este mensaje que me envió mi banco?" }`.
- **Acciones**: Enviar solicitud `POST` a `/consultar-modelo-gpt`.

- **Resultados Esperados**: Código de estado `200`, JSON con `"Calificación"` y `"Descripción"`.

### 3.4 `/consultar-modelo-gpt` Endpoint sin mensaje

- **Identificador**: `test_consultar_modelo_mensaje_faltante`
- **Descripción**: Validar el manejo de error cuando falta el campo `mensaje`.
- **Entradas**: `{ }`.
- **Acciones**: Enviar solicitud `POST`.
- **Resultados Esperados**: Código de estado `400`, JSON `{"error": "El campo 'mensaje' es obligatorio."}`.

# 4. Pruebas de Carga

**Propósito**: Evaluar el rendimiento del servicio bajo distintas cargas de usuario.

**Herramientas**: `Locust`.

## Escenarios de Prueba

### Escenario de Baja Carga

- **Descripción**: 10 usuarios concurrentes.
- **Resultados Esperados**: Tiempos de respuesta óptimos.

### Escenario de Carga Promedio

- **Descripción**: 50 usuarios concurrentes.
- **Resultados Esperados**: Respuesta adecuada sin demoras significativas.

### Escenario de Carga Máxima

- **Descripción**: 100 usuarios concurrentes.
- **Resultados Esperados**: La API debe seguir respondiendo sin fallos importantes.

## Métricas

- **Tiempo de Respuesta**: Promedio y máximo por endpoint.
- **Tasa de Éxito**: Porcentaje de solicitudes exitosas.
- **Consumo de Recursos**: CPU, memoria.

**Criterios de Aceptación**:

- Tiempos de respuesta promedio inferiores a 2 segundos en condiciones de alta carga.

# 5. Pruebas de API

**Propósito**: Validar la funcionalidad, manejo de errores y consistencia de los endpoints.

**Herramientas**: `pytest`.

### Especificación de las Pruebas de API

**`/consultar-modelo-gpt`**

- **Endpoint**: `/consultar-modelo-gpt`
- **Método HTTP**: `POST`
- **Entradas**: `{ "mensaje": "Mensaje a evaluar" }`
- **Resultados Esperados**: JSON con `Calificación` y `Descripción`.

**`/conclusion-modelo-gpt`**

- **Endpoint**: `/conclusion-modelo-gpt`
- **Método HTTP**: `POST`
- **Entradas**: `{ "resultado_parcial": "Resultado parcial" }`
- **Resultados Esperados**: JSON con `conclusion`.

# 6. Pruebas de Integración

**Propósito**: Asegurar la correcta integración entre los endpoints y el cliente de OpenAI.

**Herramientas**: `pytest` y `unittest`.

### Estrategia de Pruebas de Integración

- **Módulos Involucrados**: API de Flask, cliente OpenAI.
- **Flujos de Trabajo**: Simulación de mensajes para evaluación de riesgo.

### Casos de Prueba de Integración

- **Prueba de flujo completo en `/consultar-modelo-gpt` y `/conclusion-modelo-gpt`**:
  - **Entradas**: Mensajes de texto con características sospechosas.
  - **Resultados Esperados**: Respuestas JSON que incluyan `Calificación` o `conclusion`.

# 8. Anexos

- **Archivos de Configuración de Pruebas**: Scripts de configuración de `Locust` y `pytest`.