UNIVERSIDAD DE MONTERREY

Reporte Guiado en Equipo - Pruebas locust

MATERIA: Integración de Aplicaciones Computacionales

Eduardo Garza Briceño - 611441

Jorge Enrique Serangelli Andrade - 596711

Juan Carlos Mendoza Castillo - 598701

Maestro:

Raul Morales Salcedo

"Damos nuestra palabra que hemos realizado esta actividad con integridad académica"

Introducción

Este ejercicio consolida en un mismo reporte los resultados de tres sistemas: un CMS administrativo (HeartGuard), un microservicio de autenticación y un microservicio de catálogo de libros.

HeartGuard CMS es un panel web para superadministradores que centraliza la gestión de usuarios, organizaciones clínicas, alertas y contenidos informativos.

El **Microservicio de Catálogo de Libros** actúa como el bibliotecario central del sistema. Es un servicio especializado cuya única responsabilidad es gestionar toda la información y la lógica de negocio del catálogo. A través de su API, permite a los usuarios autorizados consultar la lista completa de libros, buscar títulos por autor o ISBN, y realizar operaciones de gestión de inventario.

El **microservicio de autenticación** es un componente de software especializado y autónomo cuya única responsabilidad es gestionar la identidad y la seguridad de los usuarios en una aplicación más grande.

Las pruebas de rendimiento permiten documentar la capacidad individual del CMS, del servicio Auth (inicio de sesión, emisión de tokens) y del servicio Libros (consulta de catálogos y guías) sin asumir dependencias entre ellos.

Importancia de las pruebas de rendimiento

Realizar pruebas de rendimiento en cualquier plataforma independiente permite:

- Identificar cuellos de botella antes de que impacten a usuarios reales.
- Dimensionar adecuadamente la infraestructura y presupuestos de capacidad.
- Evaluar la resiliencia de los mecanismos de autenticación y seguridad frente a ráfagas.
- Establecer umbrales de monitoreo y alertas basados en datos observados.
- Reducir el riesgo de degradaciones silenciosas y caídas en los procesos de negocio.

Objetivos específicos de las pruebas ejecutadas

- **Baseline**: confirmar que los flujos principales operan con latencias estables bajo carga ligera.
- **Smoke**: verificar rápidamente la disponibilidad de extremo a extremo tras despliegues o cambios en cualquier servicio.
- **Read-heavy**: medir el comportamiento cuando predominan operaciones de lectura sobre los catálogos y dashboards.
- **Write-heavy**: validar que las operaciones POST idempotentes (actualizaciones, confirmaciones) responden consistentemente bajo concurrencia moderada.
- **Ramp**: observar la degradación gradual cuando la carga crece y decrece, identificando el punto en el que surgen fallos o latencias elevadas.
- **Spike**: simular picos súbitos de tráfico para comprobar la elasticidad y la resistencia de los límites de seguridad.
- **Soak**: evaluar estabilidad sostenida en el tiempo y detectar fugas de recursos o degradaciones acumulativas.
- **Break-point**: determinar el umbral máximo de usuarios que la plataforma puede soportar antes de empezar a rechazar solicitudes.

HeartGuard CMS: Informe de Resultados de Pruebas de Carga (Eduardo Garza)

Resumen Ejecutivo

Se ejecutaron una serie de pruebas de carga sobre el CMS HeartGuard para evaluar su rendimiento, estabilidad y límites operativos bajo diversas condiciones de tráfico simulado. Los resultados confirman que el sistema maneja cargas moderadas y sostenidas con alta eficiencia, manteniendo tiempos de respuesta promedio por debajo de los 20 ms y sin registrar fallos.

Sin embargo, los escenarios de estrés (`Spike` y `Break-point`) revelaron un claro umbral de concurrencia de aproximadamente **40 usuarios simultáneos**. Superado este punto, se activa el middleware de limitación de tasa (rate-limiting), lo que provoca un aumento significativo de peticiones fallidas (respuestas HTTP 429) y una degradación severa de la latencia, particularmente en endpoints críticos como `/login` y `/superadmin/dashboard`.

Entorno y Configuración de las Pruebas

Sistema Bajo Prueba (SUT):

- **Endpoint**: `http://127.0.0.1:8080`
- **Arquitectura**: El backend consiste en una aplicación de renderizado en el servidor (SSR), una caché en Redis y una base de datos PostgreSQL.
- **Despliegue**: El entorno se ejecutó localmente utilizando los comandos `make up` y `make dev`.

Herramienta de Pruebas:

- **Software**: Locust 2.41.6
- **Dependencias Adicionales**: `zope.event` y `zope.interface`.

Datos de Prueba:

- **Usuarios**: Se utilizaron 300 credenciales de usuarios sintéticos (`loadtesterXXX@heartguard.dev`) desde un archivo `users.csv`.
- **Estado**: Todos los usuarios fueron pre-registrados en la base de datos con el rol `superadmin` para asegurar un estado consistente y realista en los escenarios de prueba autenticados.

Verificación de Salud (Health Check):

- La disponibilidad del servicio se confirmó antes de cada prueba con el comando `curl -f http://127.0.0.1:8080/health`

Resultados de los Escenarios Ejecutados

La siguiente tabla resume las métricas clave obtenidas en cada escenario de prueba.

Escenario	Descripción Breve	Usuarios / Tasa de Spawn	Duración	Peticiones	Fallos (%)	Prom. ms	P95 ms	P99 ms
Baseline	Carga ligera y sostenida para validar navegación típica.	25 / 5 s^{-1}	2m 21s	1,827	**0 (0.0%)**	13.8	28	190
Smoke	Chequeo rápido de disponibilidad post-despliegue.	5 / 2 s^{-1}	1m 50s	206	**0 (0.0%)**	12.1	27	83
Read-heavy	Mayoría de tráfico en lecturas del dashboard y listados.	60 / 6 s^{-1}	2m 10s	4,011	**0 (0.0%)**	16.5	39	230

Write-heavy	Válida POSTs idempotentes bajo concurrencia media.	40 / 4 s^{-1}	2m 44s	3,331	**0 (0.0%)**	12.2	24	150
Ramp	Escalada progresiva (20→120→20) para observar degradación.	Forma	4m 34s	8,321	**12 (0.1%)**	21.5	62	300
Spike	Pico de tráfico abrupto (15→150→15) para simular un evento.	Forma	1m 10s	3,300	**205 (6.2%)**	211.8	1,400	2,100
Soak (10 min)	Carga moderada y prolongada para detectar fugas o degradación.	40 constantes	10m 21s	12,323	**0 (0.0%)**	12.7	25	33
Break-point	Incremento de carga (+40 u/min) para encontrar el límite de fallo.	Forma	7m 48s	41,088	**3,998 (9.7%)**	53.0	120	1,200

Hallazgos Clave

1. Rendimiento Bajo Carga Normal

- **Excelente Estabilidad**: En los escenarios `Baseline`, `Read-heavy`, `Write-heavy` y `Soak`, la aplicación demostró una estabilidad excepcional, procesando todas las peticiones con **cero fallos**.
- **Baja Latencia**: Los tiempos de respuesta fueron consistentemente bajos, con promedios inferiores a 17 ms y latencias en el percentil 95 (P95) por debajo de 40 ms. Esto indica una experiencia de usuario altamente responsiva bajo condiciones operativas esperadas.

2. Límites de Concurrencia y Degradación del Servicio

- **Umbral de Rate-Limiting**: Las pruebas `Spike` y `Break-point` identificaron con éxito un techo de rendimiento claro en aproximadamente **40 usuarios concurrentes**. Superado este umbral, el sistema comienza a rechazar peticiones activamente con errores `HTTP 429 Too Many Requests`.
- **Degradación Severa del Rendimiento**: Durante el pico de 150 usuarios de la prueba `Spike`, la latencia se disparó (P99 > 2 segundos) y la tasa de fallos alcanzó el 6.2%. Esto afectó principalmente a endpoints críticos como `/login`, `/superadmin/dashboard` y `/superadmin/users`.
- **Punto de Quiebre Confirmado**: La prueba `Break-point` reforzó estos hallazgos. El endpoint `GET /superadmin/dashboard` fue el más afectado, acumulando **3,864 del total de 3,998 fallos** (una tasa de fallo global del 9.7%).

3. Autenticación y Gestión de Estado

- **Sesiones de Usuario Realistas**: El uso de credenciales únicas para cada usuario virtual, quienes mantuvieron su estado correctamente (ej. tokens `_csrf`), valida que los sistemas de autenticación y gestión de sesiones funcionan de manera correcta bajo carga.

Componentes Clave de Pruebas de Carga (Load Testing)

A continuación, se describen los componentes fundamentales implementados para la ejecución de pruebas de carga en el proyecto HeartGuard, utilizando el framework Locust. Estos componentes permiten la gestión de datos de prueba y la definición de escenarios de carga dinámicos y reutilizables.

1. Alimentador de Datos CSV (CSV Feeder)

Para simular un comportamiento realista donde cada usuario virtual utiliza credenciales únicas, se implementó un sistema de alimentación de datos (feeder) que precarga la información desde un archivo users.csv.

Descripción del Funcionamiento

El mecanismo utiliza una cola (queue.Queue) para almacenar en memoria los pares de email y contraseña antes de que comience la prueba. El evento test_start de Locust invoca la función _load_credentials() para asegurar que la cola esté poblada. Durante la ejecución, cada hilo de usuario virtual (worker) extrae un par de credenciales de la cola mediante una operación no bloqueante (get_nowait()), garantizando que no se reutilicen credenciales y evitando la contención entre hilos.

Implementación

El siguiente fragmento de código gestiona la carga y distribución de credenciales.

Python

```python
import queue
from pathlib import Path
from locust import events

# Ruta al archivo de credenciales y declaración de la cola
global
USERS_CSV = Path(__file__).resolve().parent / "data" /
"users.csv"
CREDENTIAL_POOL: queue.Queue[tuple[str, str]] = queue.Queue()

def _load_credentials() -> None:
    """
    Carga las credenciales desde el archivo CSV a la cola en
memoria.
    Se ejecuta una sola vez al inicio del test.
    """
    if CREDENTIAL_POOL.qsize() > 0:
        return
```

```
    if not USERS_CSV.exists():
        raise FileNotFoundError(f"El archivo users.csv no fue
encontrado en {USERS_CSV}")

    with USERS_CSV.open(newline="", encoding="utf-8") as
handle:
        next(handle)  # Omitir la cabecera del CSV
        for line in handle:
            email, password = line.strip().split(",",
maxsplit=1)
            if email and password:
                CREDENTIAL_POOL.put((email, password))

    if CREDENTIAL_POOL.qsize() == 0:
        raise ValueError("El archivo users.csv no contiene
credenciales válidas.")

# Hook para ejecutar la carga de credenciales al iniciar la
prueba
@events.test_start.add_listener
def on_test_start(environment, **kwargs):
    _load_credentials()

# En el método on_start de la clase User, se obtendrían así:
# self.email, self.password = CREDENTIAL_POOL.get_nowait()
```

---

2. Formas de Carga (Load Test Shapes)

Las formas de carga (LoadTestShape) son clases personalizadas que definen cómo varía el número de usuarios y la tasa de generación (spawn_rate) a lo largo del tiempo. Esta abstracción permite modelar diferentes escenarios de prueba de manera declarativa y reutilizable.

Descripción del Funcionamiento

Se implementaron dos funciones generadoras (factories):

1. _stage_shape: Crea escenarios basados en etapas, donde cada etapa tiene una duración, un número de usuarios objetivo y una tasa de generación. Es ideal para pruebas de rampa, picos y de resistencia.
2. _breakpoint_shape: Genera un escenario de carga escalonada que incrementa el número de usuarios en intervalos fijos hasta alcanzar un máximo predefinido. Su objetivo es identificar el "punto de quiebre" del sistema.

Una variable de entorno (HEARTGUARD_SHAPE) permite seleccionar la forma de carga

deseada al ejecutar Locust.

Implementación

El código a continuación define las fábricas y los escenarios de prueba disponibles.

```python
from locust.shape import LoadTestShape

def _stage_shape(name: str, stages: tuple[tuple[int, int, int], ...]) ->
type[LoadTestShape]:
    """Fábrica para crear una forma de carga basada en etapas."""
    class StageShape(LoadTestShape):
        _stages = stages

        def tick(self):
            run_time = self.get_run_time()
            elapsed = 0
            for duration, users, spawn_rate in self._stages:
                elapsed += duration
                if run_time <= elapsed:
                    return (users, spawn_rate)
            return None

    StageShape.__name__ = name
    return StageShape

def _breakpoint_shape() -> type[LoadTestShape]:
    """Fábrica para una forma de carga de tipo 'breakpoint'."""
    class BreakpointShape(LoadTestShape):
        step_duration = 60  # Duración de cada escalón en segundos
        max_users = 500     # Límite máximo de usuarios
        increment = 40      # Usuarios a agregar en cada escalón

        def tick(self):
            run_time = self.get_run_time()
            current_step = int(run_time // self.step_duration)
            users = min((current_step + 1) * self.increment, self.max_users)

            if users <= 0:
                users = self.increment

            return (users, self.increment)

    return BreakpointShape

# Diccionario de fábricas para seleccionar la forma de carga
SHAPE_FACTORIES = {
    # Carga base sostenida
    "baseline": lambda: _stage_shape("BaselineShape", ((60, 25, 5), (600, 25, 5))),
    # Incremento y decremento gradual de usuarios
    "ramp": lambda: _stage_shape(
        "RampUpDownShape",
```

```
        (
            (60, 20, 4), (180, 60, 6), (300, 120, 10),
            (420, 60, 6), (540, 20, 4), (600, 0, 1),
        ),
    ),
    # Pico de carga súbito y corto
    "spike": lambda: _stage_shape(
        "SpikeShape",
        ((30, 15, 5), (90, 150, 50), (150, 15, 5), (210, 0, 1)),
    ),
    # Carga moderada y extendida en el tiempo (30 min)
    "soak": lambda: _stage_shape("SoakShape", ((60 * 30, 40, 2),)),
    # Búsqueda del punto de quiebre
    "breakpoint": _breakpoint_shape,
}
```

Recomendaciones

El CMS HeartGuard está bien optimizado para su carga de trabajo diaria esperada. Sin embargo, la configuración actual del rate-limiter representa un cuello de botella significativo que impide al sistema gestionar picos de tráfico repentinos o escalar para acomodar un crecimiento futuro.

Se recomiendan las siguientes acciones:

1. **Revisar la Configuración del Rate-Limiter**: Se debe reevaluar la configuración del middleware en `internal/middleware/ratelimit.go`. El objetivo es encontrar un nuevo balance que continúe protegiendo contra abusos sin penalizar el tráfico legítimo durante eventos de alta demanda.
2. **Definir Umbrales de Capacidad**: El límite identificado de **40 usuarios concurrentes** debe ser utilizado como línea base para la planificación de capacidad y las reglas de autoescalado.
3. **Implementar Alertas Proactivas**: Configurar alertas de monitoreo que se disparen cuando la tasa de respuestas `HTTP 429` exceda un umbral definido. Esto permitirá al equipo abordar la degradación del rendimiento antes de que afecte de manera generalizada a los usuarios.
4. **Optimizar Endpoints de Alto Tráfico**: Investigar más a fondo el rendimiento de los endpoints `/superadmin/dashboard`, `/superadmin/users` y `/login`. Optimizaciones potenciales podrían incluir estrategias de caché más agresivas o refinar consultas a la base de datos para reducir la carga.

Pruebas de Carga y Rendimiento: Microservicio de Autenticación (Juan Carlos Mendoza)

Resumen Ejecutivo

Este informe detalla los resultados de una serie de pruebas de carga ejecutadas sobre el microservicio de Autenticación (`auth`). El objetivo principal fue diagnosticar problemas de rendimiento, validar la estabilidad tras las optimizaciones y determinar los límites operativos del sistema bajo diversas condiciones de tráfico simulado.

El hallazgo inicial fue un cuello de botella crítico y severo en el endpoint `/auth/login`, que presentaba tiempos de respuesta promedio superiores a los **10,000 ms (10 segundos)** y causaba fallos en cascada en otras partes del servicio. Tras un análisis, se formuló la hipótesis de que la causa raíz era la ausencia de índices en las columnas `username` y `email` de la base de datos, lo que resultaba en búsquedas de tabla completa (`full table scans`) altamente ineficientes.

Después de aplicar la **optimización de la base de datos**, el rendimiento del servicio experimentó una mejora drástica y medible. La latencia promedio del login se redujo en un **400%**, estabilizándose en ~2,500 ms bajo carga. El servicio demostró una robustez excepcional en pruebas de carga sostenida (`Soak Test`), procesando más de 400,000 peticiones durante 30 minutos con **cero fallos**.

Finalmente, las pruebas de punto de quiebre (`Break-point`) y de picos (`Spike`) establecieron que el nuevo límite operativo se encuentra en aproximadamente **200 usuarios concurrentes**. Superado este umbral, el sistema no falla catastrófica o masivamente, pero la latencia de respuesta se degrada de manera significativa, indicando que el siguiente cuello de botella es la capacidad de procesamiento de la base de datos bajo alta concurrencia.

Entorno y Metodología de Pruebas

Sistema Bajo Prueba (SUT - System Under Test)

- **Servicio:** Microservicio de Autenticación (`auth.py`).
- **Endpoint:** `http://4.246.170.83:5001`
- **Arquitectura Tecnológica:**
    - **Framework:** Flask (Python)
    - **Base de Datos:** MySQL
    - **Autenticación:** Basada en tokens JWT (JSON Web Tokens)
- **Despliegue:** Una instancia única del servicio ejecutándose en un servidor virtual.

Herramienta de Generación de Carga

- **Software:** Locust v2.41.5
- **Entorno de Ejecución:** Pruebas lanzadas en modo `--headless` desde una máquina local.

2.3. Datos y Escenario de Prueba

- **CSV Feeder:** Se utilizó un archivo `users.csv` con **300 credenciales de usuarios únicos** para simular un comportamiento de login realista y evitar el caché de consultas repetitivas.
- **Estado Inicial:** Todos los usuarios de prueba fueron pre-registrados en la base de datos.
- **Flujo de Usuario Simulado (Read-Heavy):** El comportamiento de los usuarios fue configurado para ser de lectura intensiva, con 5 peticiones `GET /api/user-profile` por cada petición `POST /auth/refresh`.

Resumen Tabular de Pruebas Ejecutadas

La siguiente tabla consolida los resultados de todos los escenarios ejecutados, diferenciando entre las pruebas de diagnóstico (pre-optimización) y las de validación (post-optimización).

Escenario	Descripción Breve	Usuarios / Tasa de Spawn	Duración	Peticiones	Fallos (%)	Prom. ms	P95 ms	P99 ms
Smoke (Pre-Opt)	Chequeo rápido que reveló el cuello de botella.	5 / 1 s^{-1}	30s	~47,800	2 (0.67 %)	1,226	~21,000	~21,000
Baseline (Pre-Opt)	Confirmó la inestabilidad y las fallas en cascada.	50 / 5 s^{-1}	5m	~10,000	10 (0.10 %)	708	>20,000	>50,000
Read-Heavy (Post-Opt)	Validó el rendimiento bajo tráfico de lectura intenso.	100 / 10 s^{-1}	10m	~72,000	**0 (0.00 %)**	210	350	3,100
Soak (Post-Opt)	Validó la estabilidad a largo plazo tras la optimización.	200 constantes	30m	~478,500	**0 (0.00 %)**	310	1,500	3,550

Ramp/Spike (Post-Opt)	Probó la resiliencia ante un pico de tráfico abrupto.	Forma (150→1000)	~9m	~50,000	215 (0.43 %)	950	4,800	15,000
Break-point (Post-Opt)	Encontró el nuevo límite de degradación del servicio.	500 / 50 s^{-1}	5m	~35,000	9 (0.02 %)	850	4,500	51,000

Análisis Detallado de Escenarios de Prueba

Implementación

El siguiente fragmento de código es responsable de verificar las credenciales de un usuario (email y contraseña) contra la base de datos. Si la autenticación es exitosa, genera un JSON Web Token (JWT) que se utilizará para autorizar las solicitudes posteriores del usuario a la API.

Python

```python
import jwt

import datetime

from werkzeug.security import check_password_hash

from db_connector import get_db_connection  # Módulo hipotético para la conexión a BD



# Clave secreta para firmar el token, debe cargarse desde un lugar seguro

SECRET_KEY = "mi-clave-ultra-secreta-de-desarrollo"



def authenticate_and_generate_token(email: str, password: str) -> dict | None:

    """

    Busca al usuario por email, verifica su contraseña y, si es correcta,

    genera y devuelve un token de acceso JWT.



    Returns:
```

```
        Un diccionario con el token o None si las credenciales
son inválidas.
    """
    if not email or not password:
        return None

    conn = get_db_connection()
    try:
        with conn.cursor() as cursor:
            # 1. Buscar al usuario y su hash de contraseña en
la base de datos
            cursor.execute(
                "SELECT id, password_hash, user_status_id FROM
heartguard.users WHERE email = %s",
                (email.lower(),)
            )
            user_record = cursor.fetchone()

            if not user_record:
                return None  # Usuario no encontrado

            user_id, stored_hash, status_id = user_record
```

```
            # 2. Verificar que la contraseña proporcionada
coincida con el hash
            if not check_password_hash(stored_hash, password):
                return None  # Contraseña incorrecta

            # 3. Crear el payload (contenido) del token
            payload = {
                "sub": str(user_id),  # 'subject' del token
(identificador único del usuario)
                "status": str(status_id), # Se puede incluir
información adicional como el estado
                "iat": datetime.datetime.utcnow(),  # 'issued
at time' (hora de emisión)
                "exp": datetime.datetime.utcnow() +
datetime.timedelta(hours=24) # 'expiration time'
            }

            # 4. Generar el token JWT firmado con la clave
secreta
            token = jwt.encode(payload, SECRET_KEY,
algorithm="HS256")

            return {"access_token": token}
```

```
    finally:
        conn.close()


# Ejemplo de cómo se usaría en una ruta de login (ej. en un framework como Flask)
#
# @app.route("/api/v1/login", methods=["POST"])
# def login():
#     data = request.get_json()
#     tokens = authenticate_and_generate_token(data.get("email"), data.get("password"))
#     if tokens:
#         return jsonify(tokens)
#     else:
#         return "Credenciales inválidas", 401
```

Fase 1: Diagnóstico (Pre-Optimización)

En esta fase, el objetivo era identificar los cuellos de botella del sistema en su estado original.

- **Prueba Smoke:** El chequeo inicial con 5 usuarios fue suficiente para detectar el problema. La latencia de **11 segundos** en el login y el impacto colateral en el endpoint `/api/user-profile` (picos de 19 segundos) señalaron un problema grave en la base de datos.
- **Prueba Baseline:** Al aumentar la carga a 50 usuarios, el problema se agravó. El endpoint `/api/user-profile`, que antes no fallaba, comenzó a registrar errores

(9 fallos), confirmando un patrón de **falla en cascada** originado por la lentitud del login.

Fase 2: Validación (Post-Optimización)

Después de aplicar los índices en la base de datos, se ejecutó una serie completa de pruebas para validar la solución.

- **Prueba Read-Heavy:** Con 100 usuarios concurrentes, el sistema demostró un rendimiento excelente, con una latencia promedio de solo 210 ms y cero fallos. Esto confirmó que, una vez resuelto el problema del login, las operaciones de lectura eran extremadamente eficientes.
- **Prueba Soak:** Este fue el test más importante para validar la estabilidad. Soportar 200 usuarios durante 30 minutos sin un solo error y con tiempos de respuesta bajos y consistentes demostró que la solución era robusta y no había fugas de memoria o recursos.
- **Prueba Ramp-up / Spike:** Esta prueba simuló un evento de tráfico repentino. El sistema manejó bien el crecimiento gradual (ramp-up) hasta los ~200 usuarios. Sin embargo, durante el pico abrupto a 1000 usuarios, la latencia P99 se disparó a **15 segundos** y se registraron algunos fallos. Esto prueba que el sistema es resiliente (no colapsa), pero la experiencia del usuario se degrada severamente por encima del umbral de ~200 usuarios.
- **Prueba Break-point:** Este escenario confirmó los hallazgos de la prueba Spike. Al intentar escalar a 500 usuarios, el sistema no se rompió, sino que se "dobló", encolando peticiones hasta que la latencia P99 alcanzó **51 segundos**. Esto establece de manera definitiva que el límite operativo para una experiencia de usuario aceptable está en la marca de los 200 usuarios concurrentes.

Recomendaciones

El microservicio de Autenticación ahora es **estable y apto para producción** para una carga de trabajo de hasta 200 usuarios concurrentes. Para escalar más allá de este punto, se recomiendan las siguientes acciones:

1. **Acción Inmediata (Monitoreo):**
    - **Implementar Alertas Basadas en Latencia:** Configurar alertas que se disparen cuando la latencia **P95 o P99** exceda un umbral predefinido (ej. 3 segundos). Esto detectará la degradación del servicio antes de que los usuarios experimenten fallos.
2. **Acción a Medio Plazo (Optimización y Escalabilidad):**
    - **Implementar un Pool de Conexiones a la BD:** Optimizar SQLAlchemy para usar un pool de conexiones persistente (`QueuePool`) para reducir la sobrecarga de crear nuevas conexiones bajo alta concurrencia.
    - **Introducir Caching para Datos de Usuario:** Implementar una caché en memoria (Redis) para el endpoint `GET /api/user-profile` podría descargar una cantidad masiva de tráfico de lectura de la base de datos.
    - **Escalado Horizontal:** Para manejar más de 200 usuarios, desplegar múltiples instancias del servicio detrás de un balanceador de carga.
3. **Acción a Largo Plazo (Investigación):**
    - **Analizar el Rendimiento de Bcrypt:** El hashing de contraseñas es intensivo en CPU. Investigar si una implementación asíncrona podría ofrecer mejoras si el login sigue siendo el endpoint más lento.

Conclusión

El proceso de pruebas de carga fue exitoso, no solo al identificar un problema crítico de rendimiento, sino también al validar la efectividad de la solución implementada. El microservicio de autenticación ha pasado de ser inestable a un estado **robusto, predecible y listo para operar en un entorno de producción** dentro de sus límites de capacidad ahora conocidos. Se ha establecido una base sólida y un camino claro para futuras optimizaciones y escalado.

Reporte Final de Pruebas de Carga: Microservicio Libros (Jorge Serangelli)

Introducción

El microservicio de Libros es un componente de software especializado y autónomo cuya única responsabilidad es gestionar toda la información y la lógica de negocio relacionada con el catálogo de libros de la aplicación.

En términos simples, actúa como el **"bibliotecario central"** del sistema. Sus tareas principales son:

- **Exponer el Catálogo:** Provee una lista completa y detallada de todos los libros disponibles, incluyendo su título, autor, género y precio.
- **Encontrar Libros Específicos:** Permite a los usuarios y a otros servicios realizar búsquedas precisas, ya sea por ISBN, por autor, por formato o por otros criterios.
- **Gestionar el Inventario (CRUD):** Se encarga de todas las operaciones del ciclo de vida de un libro: agregar nuevos títulos al sistema (`Create`), leer su información (`Read`), actualizar datos como el stock o el precio (`Update`) y eliminar libros del catálogo (`Delete`).
- **Servir Datos a Peticiones Autorizadas:** Responde únicamente a las peticiones de usuarios que han sido validados previamente por el microservicio de Autenticación. No entrega información a nadie que no presente una "credencial" (token JWT) válida.

En resumen, centraliza toda la lógica de negocio del catálogo para que el resto de la aplicación no necesite saber cómo se almacenan o gestionan los libros. Otros servicios simplemente le preguntan a este: "¿Puedes darme los detalles de este libro?" o "¿Puedes agregar este nuevo título al inventario?".

Resumen Ejecutivo

Este informe detalla los resultados de una serie de pruebas de carga ejecutadas sobre el **microservicio de Libros**. El objetivo principal fue diagnosticar y resolver los cuellos de botella de rendimiento, validar la estabilidad del sistema bajo diversas condiciones de carga y determinar sus límites operativos.

El sistema en su estado inicial era **completamente inestable** y no soportaba la más mínima concurrencia. Las pruebas iniciales revelaron fallos catastróficos en cascada, incluyendo `ConnectionResetError`, `Too many connections`, y `pool exhausted`, que hacían que la aplicación fuera inviable. La causa raíz fue identificada como una arquitectura de desarrollo (servidor de Flask y gestión de conexiones a la base de datos inadecuada) que no estaba preparada para un entorno de producción.

Tras un proceso iterativo de optimización, se implementó una **arquitectura de producción robusta** basada en **Gunicorn** y un **pool de conexiones a la base de datos "auto-reparable"**. La validación posterior demostró una mejora drástica y medible:

- **Estabilidad Absoluta:** El sistema finalizó una prueba de carga sostenida (Soak Test) de 30 minutos con **cero fallos**.
- **Resiliencia Comprobada:** Soportó un pico de tráfico de **1,000 usuarios** (Spike Test) con una tasa de fallos insignificante (< 0.1%) y se recuperó sin problemas.
- **Límite de Rendimiento Definido:** La prueba de punto de quiebre (Break-point) a **800 usuarios** no rompió el sistema, sino que reveló su límite por degradación de rendimiento. La latencia se disparó a una mediana de **22 segundos**, demostrando que la arquitectura es estable pero que los recursos de la máquina virtual se saturan a ese nivel.

En conclusión, el microservicio de Libros pasó de ser un sistema frágil a una aplicación robusta, escalable y resiliente, apta para un entorno de producción con una capacidad operativa claramente definida.

Entorno y Datos de Prueba

- **Infraestructura:** Google Cloud Platform (GCP)
- **Servidor de Aplicación:** Gunicorn
    - **Servicio Auth:** 2 workers (`--workers 2`)
    - **Servicio Libros:** 4 workers (`--workers 4`)
- **Microservicios:**
    - **Auth (Puerto 5000):** Basado en Flask, responsable del registro y login.
    - **Libros (Puerto 5001):** Basado en Flask, expone la API del catálogo.
- **Base de Datos:** MariaDB
- **Cache:** Redis (para gestión de tokens JWT)
- **Generador de Carga:** Locust, ejecutado desde una máquina local.
- **Datos de Prueba:** 300 usuarios únicos generados y pre-registrados en la base de datos (`users.csv`).

Escenarios Ejecutados y Análisis de Resultados

Se ejecutaron 8 escenarios de prueba distintos. La arquitectura final demostró ser **altamente estable**, con una tasa de éxito superior al **99.5%** en las pruebas más intensivas, validando la robustez del sistema.

#	Prueba	Usuarios Máx.	Duración	Peticiones/s (RPS)	Tpo. Respuesta (Mediana ms)	Fallos (%)	Estado
1	**Smoke Test**	5	1 min	1.9	150ms	**(0.00%)**	**Éxito**
2	**Baseline**	50	5 min	21	200ms	**(0.00%)**	**Éxito**
3	**Read-Heavy**	100	5 min	45	230ms	**(0.00%)**	**Éxito**
4	**Write-Heavy**	100	5 min	40	280ms	**(0.00%)**	**Éxito**
5	**Stages (Rampas)**	500	~7 min	183	360ms	(**0.04%**)	**Éxito**
6	**Soak (Sostenida)**	150	30 min	62	250ms	**(0.00%)**	**Éxito**
7	**Break-Point**	800	Manual	3.17	**22,000ms**	(**0.45%**)	**Límite Encontrado**

8	**Spike (Pico)**	1,000	~3 min	291	440ms	(**0.08%**)	**Éxito**

Hallazgos Clave y Análisis a Profundidad

El proceso de pruebas fue un ciclo iterativo de "romper y arreglar" que expuso debilidades críticas en la arquitectura inicial. Los hallazgos no fueron solo números, sino una guía para la evolución del sistema.

Hallazgo 1: El Servidor de Desarrollo es un Punto de Falla Catastrófico

Las pruebas iniciales ni siquiera pudieron arrancar. Locust reportó errores masivos de `ConnectionResetError` y `ConnectTimeoutError`.

- **Análisis a Profundidad:** El problema no estaba en el código de la aplicación, sino en su base: el **servidor de desarrollo de Flask (`app.run`)**. Este servidor es de un solo hilo y no está diseñado para peticiones concurrentes. Al recibir la carga de Locust, simplemente se colapsaba, reiniciando o rechazando conexiones antes de que pudieran ser procesadas.
- **Solución Implementada:** Se migró la ejecución de ambos servicios a **Gunicorn**, un servidor WSGI de producción. Al configurar Gunicorn con múltiples workers, se habilitó el procesamiento en paralelo real, estabilizando la capa de aplicación.

Hallazgo 2: La Estrategia de Conexión a la BD es Crítica

Una vez con Gunicorn, la aplicación se mantenía en pie, pero la base de datos se convirtió en el siguiente cuello de botella. El proceso de depuración reveló tres etapas de este problema:

1. `Too many connections`: El patrón de "una conexión por petición" agotó rápidamente el límite de conexiones de MariaDB. **Esto demostró que la base de datos era el principal cuello de botella de la arquitectura.**
2. `Pool exhausted`: La implementación de un pool de conexiones ayudó, pero inicialmente era demasiado pequeño (`pool_size=15`) y se agotaba, causando fallos.
3. **Errores 500 Intermitentes (Conexiones "Stale")**: Incluso con un pool más grande, aparecieron errores `500` esporádicos. El análisis reveló que la base de datos cerraba conexiones que permanecían inactivas en el pool por mucho tiempo.

- **Solución Implementada:** Se implementó un **pool de conexiones "auto-reparable"** en **ambos** microservicios usando `mysql-connector-python`, con un tamaño de 32 y el parámetro `pool_reset_session=True`. Esto aseguró que la aplicación nunca intentara abrir más conexiones de las permitidas y que las conexiones "zombies" fueran reemplazadas automáticamente.

Hallazgo 3: El Punto de Quiebre es por Degradación, no por Errores

La prueba de Break-Point con 800 usuarios fue el hallazgo más revelador, basado en el archivo `7_breakpoint_report_stats.csv`.

- **Análisis a Profundidad:** Contrario a lo esperado, el sistema no falló con errores masivos (solo **5 fallos** de un total de 1,123 peticiones, una tasa de error de apenas **0.45%**). En su lugar, sufrió una **degradación de rendimiento catastrófica**. Los tiempos de respuesta se dispararon a una mediana de **22 segundos** (22,000 ms) y un promedio de **26 segundos**. El rendimiento (RPS) se desplomó a solo **3.17 peticiones por segundo**. Esto indica que la arquitectura es muy **estable** (no se cae), pero que los recursos del servidor (CPU/memoria) se saturaron por completo.
- **Conclusión del Hallazgo:** El límite del sistema no se define por errores, sino por una latencia que hace la aplicación inutilizable. Este límite se encuentra alrededor de los 800 usuarios concurrentes con la infraestructura actual.

Hallazgo 4: Resiliencia y Estabilidad Comprobadas

Las pruebas de **Soak** y **Spike** validaron la robustez de la arquitectura final.

- **Prueba Sostenida (Soak Test):** El sistema manejó 150 usuarios durante 30 minutos sin un solo error y con tiempos de respuesta bajos y constantes (mediana de 250 ms). Esto confirma que no hay fugas de memoria ni degradación del rendimiento a largo plazo.
- **Prueba de Pico (Spike Test):** El sistema absorbió un pico repentino de 1,000 usuarios con una tasa de fallos mínima (solo 20 fallos, < 0.1%) y se recuperó sin problemas. Esto demuestra una gran elasticidad y capacidad para manejar eventos de tráfico inesperados.

El sistema, en su estado actual, es **robusto, escalable y resiliente**. Las pruebas de carga fueron un éxito rotundo, no solo al validar el rendimiento, sino al **forzar la evolución de una arquitectura de desarrollo frágil a una arquitectura de producción sólida**. El punto de quiebre, identificado por una degradación severa del rendimiento a los 800 usuarios, demuestra una alta capacidad para manejar picos de tráfico, manteniendo la integridad del sistema incluso bajo estrés extremo.

Componentes Clave de Pruebas de Carga (Load Testing)

A continuación, se describen los componentes fundamentales implementados para la ejecución de pruebas de carga en el proyecto de Microservicios, utilizando el framework Locust. Estos componentes permiten la gestión de datos de prueba, la definición del comportamiento del usuario y la creación de escenarios de carga dinámicos.

Alimentador de Datos CSV (CSV Feeder)

Para simular un comportamiento realista donde cada usuario virtual utiliza credenciales únicas, se implementó un sistema de alimentación de datos (feeder) que lee la información desde un archivo users.csv.

Descripción del Funcionamiento

El mecanismo implementado es simple y altamente eficiente. Al iniciar el script de Locust, se leen todas las credenciales del archivo users.csv y se cargan en una lista en memoria. Posteriormente, se utiliza la función itertools.cycle para crear un iterador infinito sobre esa lista.

Durante la ejecución de la prueba, cada vez que un nuevo usuario virtual es creado, simplemente toma el siguiente par de credenciales de este iterador (next(user_credentials)). Si se llega al final de la lista, el iterador vuelve a empezar desde el principio automáticamente. Este enfoque es simple, seguro para hilos (thread-safe) y garantiza una distribución continua de usuarios únicos.

Implementación

El siguiente código, extraído de **locustfile.py**, gestiona la carga y distribución de credenciales a cada usuario virtual.

Python

```
None
import csv

import itertools

import random

import uuid

from locust import HttpUser, task, between, LoadTestShape

import time
```

```
# --- CONFIGURACIÓN ---

AUTH_SERVICE_URL = "http://35.225.153.19:5000"

BOOKS_SERVICE_URL = "http://35.225.153.19:5001"


# --- CSV FEEDER ---

try:

    with open('users.csv', 'r') as f:

        user_credentials =
itertools.cycle(list(csv.DictReader(f)))

except FileNotFoundError:

    print("Error: El archivo 'users.csv' no fue encontrado.
Por favor, genéralo primero.")

    exit()


class AuthenticatedUser(HttpUser):

    """

    Clase base que maneja la autenticación para cada usuario
virtual.

    """

    abstract = True

    token = None


    def on_start(self):

        self.login()
```

```
    def login(self):
        """Obtiene credenciales del CSV y realiza el login
enviando el username."""
        credentials = next(user_credentials)
        try:
            response =
self.client.post(f"{AUTH_SERVICE_URL}/login", json={
                "username": credentials['username'],
                "password": credentials['password']
            }, name="/login")

            if response.status_code == 200:
                self.token =
response.json().get('access_token')
            else:
                self.token = None
        except Exception:
            self.token = None

    def get_auth_headers(self):
        if not self.token:
            return {}
        return {'Authorization': f'Bearer {self.token}'}
```

Comportamiento del Usuario (Clase BookstoreUser)

Para simular las acciones de un usuario real en el microservicio de Libros, se definió la clase BookstoreUser. Esta clase hereda la lógica de autenticación y define las tareas que cada usuario virtual ejecutará.

Descripción del Funcionamiento

Se utiliza el decorador @task de Locust para definir las diferentes acciones (peticiones a la API). A cada tarea se le asigna un "peso" numérico, que determina su probabilidad de ser ejecutada. En nuestro caso, se configuró un escenario de **lectura intensiva (Read-Heavy)**, donde las operaciones GET (consultar libros) son mucho más frecuentes que las operaciones POST (insertar un libro nuevo). Esto simula un patrón de uso más realista para una tienda de libros en línea.

Implementación

El siguiente código muestra la clase BookstoreUser completa, con sus tareas y pesos, extraída de **locustfile.py**.

Python

```
None
class BookstoreUser(AuthenticatedUser):

    """

    Define las tareas que un usuario realizará en el
microservicio de libros.

    """

    host = BOOKS_SERVICE_URL

    wait_time = between(1, 5)



    @task(10)

    def get_all_books(self):

        self.client.get("/api/books",
headers=self.get_auth_headers(), name="/api/books [GET]")



    @task(5)
```

```
    def get_book_by_isbn(self):

        isbns = ["978-0061120084", "978-0140449136",
"978-0553380163"]

        isbn = random.choice(isbns)

        self.client.get(f"/api/books/isbn/{isbn}",
headers=self.get_auth_headers(), name="/api/books/isbn/[isbn]
[GET]")



    @task(3)

    def get_books_by_author(self):

        authors = ["George Orwell", "Homer", "Jane Austen"]

        author = random.choice(authors)

        self.client.get(f"/api/books/author/{author}",
headers=self.get_auth_headers(),
name="/api/books/author/[author] [GET]")



    @task(1)

    def insert_book(self):

        genres = ["Fiction", "Epic", "Romance", "Dystopian"]

        formats = ["Físico", "Digital"]

        authors = ["George Orwell", "Jane Austen"]



        payload = {

            "isbn":
f"TEST-{int(time.time())}-{random.randint(1000, 9999)}",
```

```
            "title": f"New Test Book {random.randint(1000,
9999)}",
            "year": random.randint(1990, 2025),
            "price": round(random.uniform(9.99, 49.99), 2),
            "stock": random.randint(1, 100),
            "genre": random.choice(genres),
            "format": random.choice(formats),
            "authors": random.choice(authors)
        }
        self.client.post("/api/books/insert", json=payload,
headers=self.get_auth_headers(), name="/api/books/insert
[POST]")
```

Formas de Carga (Load Test Shapes)

Las formas de carga (LoadTestShape) son clases personalizadas que definen cómo varía el número de usuarios a lo largo del tiempo. Esto nos permite modelar escenarios complejos como rampas graduales o picos de tráfico repentinos.

Descripción del Funcionamiento

Debido a que la versión de Locust utilizada no soportaba la selección de LoadTestShape por argumento de línea de comandos, se optó por una estrategia de **archivos separados**. Cada escenario con una forma de carga se implementó en su propio archivo de Locust.

1. **StagesShape** (en locustfile.py): Define un escenario de múltiples etapas para simular rampas de subida, picos sostenidos y rampas de bajada.
2. **SpikeTestShape** (en locustfile_spike.py): Define un escenario que simula un pico de tráfico repentino y masivo.

El script run_all_tests.sh invoca el archivo de Locust correcto para cada prueba, permitiendo la ejecución automatizada de ambos perfiles de carga.

Implementación

El código a continuación muestra las dos clases LoadTestShape implementadas.

1. StagesShape para rampas y picos sostenidos (de locustfile.py):

Python

None

```
from locust import LoadTestShape


class StagesShape(LoadTestShape):
    """
    Define la carga con rampas y picos.
    """
    stages = [
        {"duration": 60, "users": 100, "spawn_rate": 10},
        {"duration": 180, "users": 100, "spawn_rate": 10},
        {"duration": 190, "users": 500, "spawn_rate": 100},
        {"duration": 240, "users": 500, "spawn_rate": 50},
        {"duration": 300, "users": 50, "spawn_rate": 50},
        {"duration": 360, "users": 50, "spawn_rate": 10},
    ]

    def tick(self):
        run_time = self.get_run_time()
        for stage in self.stages:
            if run_time < stage["duration"]:
                return (stage["users"], stage["spawn_rate"])
        return None
```

2. SpikeTestShape para picos de tráfico repentinos (de locustfile_spike.py):

Python

None

```
# Importa las clases necesarias del archivo principal para reutilizar el código

from locustfile import BookstoreUser, LoadTestShape


class SpikeTestShape(LoadTestShape):

    """

    Define una carga que simula un pico de tráfico repentino.

    """

    stages = [

        {"duration": 60, "users": 50, "spawn_rate": 10},
# 1. Baseline por 1 min

        {"duration": 70, "users": 1000, "spawn_rate": 95},
# 2. Spike: +950 usuarios en 10 seg

        {"duration": 130, "users": 1000, "spawn_rate": 50},
# 3. Hold: Mantener pico por 1 min

        {"duration": 150, "users": 0, "spawn_rate": 50},
# 4. Ramp-down a 0

    ]


    def tick(self):

        run_time = self.get_run_time()

        for stage in self.stages:

            if run_time < stage["duration"]:

                return (stage["users"], stage["spawn_rate"])

        return None
```

Recomendaciones

El microservicio de Libros, en su arquitectura final, es **robusto, resiliente y está listo para un entorno de producción**. Las pruebas demostraron que los cuellos de botella críticos a nivel de aplicación (servidor y gestión de conexiones) han sido resueltos exitosamente. El límite operativo del sistema ya no es su software, sino la capacidad de la infraestructura subyacente.

Se recomiendan las siguientes acciones para gestionar y escalar el servicio de manera efectiva:

Acción Inmediata (Monitoreo)

- **Implementar Alertas Basadas en Latencia, no en Errores:** El hallazgo más crítico de la prueba de Break-Point fue que el sistema no falla con errores, sino que sufre una **degradación severa del rendimiento**. Por lo tanto, se debe configurar un sistema de alertas que se dispare cuando la latencia P95 exceda un umbral definido (ej. **2 segundos**). Esto permitirá detectar la saturación del sistema mucho antes de que la experiencia del usuario se vea afectada.

Acción a Medio Plazo (Optimización y Escalabilidad)

- **Escalado Vertical de la Infraestructura:** El límite de ~800 usuarios se debe a la saturación de los recursos de la máquina virtual (CPU/memoria). La forma más directa de aumentar la capacidad del sistema es escalar verticalmente la instancia en GCP, asignándole más vCPUs y RAM.
- **Análisis de Consultas Lentas (Slow Queries):** Ahora que la aplicación es estable, se debe habilitar y monitorear el log de consultas lentas en MariaDB durante picos de carga para identificar si alguna consulta específica (ej. `GROUP_CONCAT`) se convierte en un cuello de botella y puede ser optimizada.

Acción a Largo Plazo (Escalabilidad Avanzada)

- **Implementar Caching para el Catálogo:** El endpoint `GET /api/books` es probablemente el más consultado. Implementar una capa de caché (utilizando Redis, que ya está en el stack) para almacenar el resultado de esta consulta durante un periodo corto (ej. 1-5 minutos) podría reducir drásticamente la carga de lectura sobre la base de datos, permitiendo al sistema manejar a más usuarios con los mismos recursos.
- **Escalado Horizontal:** Para superar significativamente el límite de 800 usuarios, el siguiente paso es desplegar **múltiples instancias** del microservicio de Libros detrás de un balanceador de carga. La arquitectura actual, al ser estable y no depender de estado local, está perfectamente preparada para este modelo de escalabilidad.

Conclusión

El proceso de pruebas de carga y rendimiento sobre el microservicio de Libros fue un éxito rotundo y fundamental. Se logró transformar una aplicación inicialmente frágil e inestable, incapaz de soportar la más mínima concurrencia, en un sistema **robusto, resiliente y preparado para un entorno de producción**.

Los hallazgos iniciales, que incluyeron fallos catastróficos como `Too many connections` y `Pool exhausted`, fueron cruciales para guiar la reingeniería de la arquitectura. La migración del servidor de desarrollo de Flask a **Gunicorn** y, de manera crítica, la implementación de un **pool de conexiones a la base de datos "auto-reparable"**, demostraron ser las soluciones definitivas que eliminaron los cuellos de botella y estabilizaron el sistema.

La arquitectura final validó su robustez en todos los escenarios de prueba:

- Demostró una **estabilidad absoluta** en la prueba sostenida (Soak Test), operando durante 30 minutos sin un solo error.
- Probó su **alta resiliencia** al absorber un pico de 1,000 usuarios (Spike Test) con una tasa de fallos insignificante y una recuperación inmediata.
- Definió un **límite operativo claro** en la prueba de punto de quiebre (Break-Point), donde el sistema no colapsó con errores, sino que mostró una degradación predecible del rendimiento, evidenciando su estabilidad incluso bajo estrés extremo.

En definitiva, las pruebas de carga no solo midieron el rendimiento, sino que **forzaron la evolución de la aplicación**, validando que la arquitectura final es la correcta. El microservicio de Libros ahora posee una capacidad conocida y fiable, listo para ser desplegado con la confianza de que puede manejar cargas de trabajo significativas y eventos de tráfico inesperados.