UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

SEDE LATACUNGA

INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Pruebas de Carga y Rendimiento con k6
Autor(a):	Juanito y Pepito
Carrera:	Ingeniería en Software
Asignatura:	Pruebas de Software
Nivel:	6to
Docente:	Ing. Enrique Calvopiña, Mgtr.
NRC:	22431
Práctica N°:	5
Fecha de entrega:	6 de enero de 2026

Índice

1. Introducción

Este informe presenta el desarrollo y los resultados obtenidos en el laboratorio de pruebas de carga y rendimiento de servicios API REST. El propósito fundamental es evaluar la capacidad de respuesta y estabilidad de una interfaz programática ante escenarios de uso intensivo mediante la simulación de usuarios concurrentes.

El aseguramiento de la calidad en el software moderno requiere no solo verificar la funcionalidad, sino también garantizar que el sistema pueda operar bajo carga sin degradar la experiencia del usuario final Sahli et al. (2023). Para este propósito, la utilización de herramientas de orquestación de carga como k6 permite realizar pruebas de rendimiento de manera programática, facilitando la identificación de cuellos de botella en arquitecturas de backend distribuidas Vuković et al. (2020).

En las siguientes secciones se detalla el proceso de configuración del ambiente, la creación de los scripts de prueba y el análisis exhaustivo de las métricas obtenidas, permitiendo identificar posibles cuellos de botella y áreas de mejora en la infraestructura del servidor.

2. Objetivos

2.1. Objetivo General

Desarrollar y ejecutar pruebas de carga y rendimiento sobre una API REST mediante el uso de la herramienta k6, con el fin de evaluar la capacidad de respuesta y estabilidad del servidor ante la simulación de múltiples usuarios virtuales concurrentes.

2.2. Objetivos Específicos

- Implementar un servidor API REST utilizando Node.js y Express para servir como entorno controlado de pruebas.
- Configurar scripts de prueba en k6 que definan umbrales de rendimiento (*thresholds*) y escenarios de carga progresiva.
- Interpretar y analizar las métricas obtenidas (latencia, throughput, tasa de errores) para determinar si el sistema cumple con los criterios de calidad establecidos.

3. Marco Teórico

3.1. Node.js y Express

Node.js es un entorno de ejecución de JavaScript orientado a eventos, diseñado para construir aplicaciones de red escalables. Express es el framework web más popular para Node.js, facilitando la creación de APIs REST mediante el manejo eficiente de rutas y middleware.

3.2. k6 y las Pruebas de Carga

k6 es una herramienta de pruebas de rendimiento de código abierto, moderna y centrada en el flujo de trabajo del desarrollador. En el contexto de arquitecturas modernas, las pruebas de carga permiten mitigar riesgos asociados a la escalabilidad y al consumo de recursos en entornos de producción saturados Bezemer et al. (2019). Esta herramienta permite escribir scripts en JavaScript, optimizando la simulación de usuarios virtuales (VUs) con un consumo eficiente de memoria.

3.3. Métricas de Rendimiento

Las métricas analizadas en este estudio incluyen el tiempo de respuesta, throughput y la tasa de peticiones fallidas, elementos críticos para determinar el Punto de Saturación de un sistema backend Vuković et al. (2020).

4. Desarrollo

A continuación, se detalla la ejecución del laboratorio dividida en tres partes fundamentales, conforme a la guía de actividades.

4.1. Parte 1: Establecimiento del ambiente de pruebas

4.1.1. Paso 1: Creación de API sencilla

Se procedió con la creación del archivo `server.js` (ver Listing 1), configurando un servidor básico con Express que incluye:

- Importación del módulo Express.
- Implementación de rutas GET para simular respuestas simples con un retardo aleatorio de hasta 500ms.
- Implementación de una ruta POST para recibir y procesar datos JSON.
- Configuración del servidor para escuchar en el puerto 3333.

```
const express = require('express');
const app = express();
app.use(express.json());

app.get('/api/test', (req, res) => {
    const sleep = Math.floor(Math.random() * 500);
    setTimeout(() => {
        res.json({ message: 'Peticion procesada' });
    }, sleep);
})

app.post('/api/data', (req, res) => {
    res.status(201).json({ received: req.body });
```

```
});

app.listen(3333, '0.0.0.0', () => {
    console.log('Servidor escuchando en http://localhost:3333');
});
```

Listing 1: Implementación del servidor API REST

4.1.2. Paso 2: Instalación de dependencias necesarias

Se gestionó el proyecto mediante `npm` siguiendo estos subpasos:

1. Inicialización del proyecto con `npm init -y`.
2. Instalación de la dependencia de Express mediante `npm install express`.
3. Verificación de la ejecución con `node server.js`.

Como se observa en la Figura 1, el servidor fue inicializado exitosamente tras la configuración de dependencias. Posteriormente, se validó el correcto funcionamiento de los endpoints mediante la herramienta `curl`; en la Figura 2 se aprecia la respuesta exitosa del método GET, mientras que en la Figura 3 se confirma que ambos endpoints (GET y POST) se encuentran operativos y respondiendo según lo diseñado.

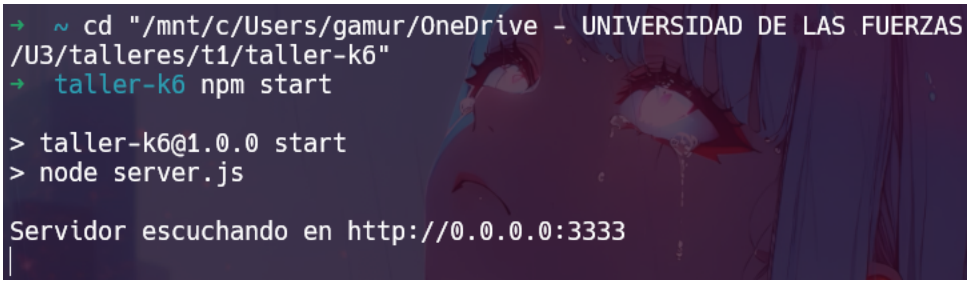

Figura 1: Creación y estructura inicial de la API en Node.js.

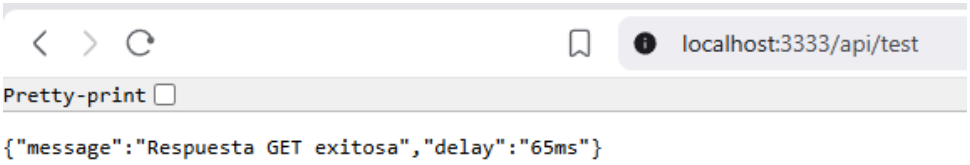

Figura 2: Prueba de funcionamiento del endpoint GET mediante curl.

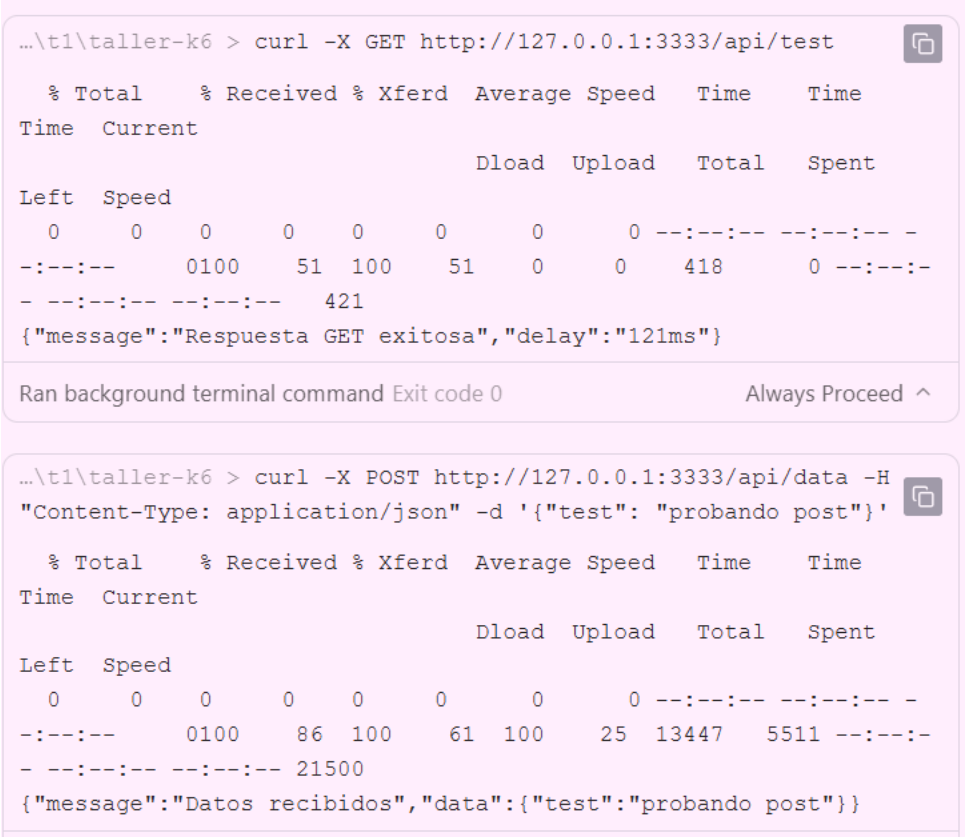

Figura 3: Validación de ambos endpoints (GET y POST) activos en el servidor.

4.1.3. Paso 3: Instalación de k6

Dado que el ambiente se basa en WSL (Ubuntu), se realizó la instalación utilizando el repositorio oficial de k6, siguiendo los comandos detallados en el Listing 2. Se verificó la autenticidad del paquete mediante la importación de la llave GPG y la configuración del archivo `k6.list` en el directorio de fuentes de apt para asegurar una instalación estable y libre de errores de dependencias.

```
# 1. Importar la llave GPG de k6
sudo gpg --no-default-keyring --keyring /usr/share/keyrings/k6-archive-keyring.gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys C5AD17C747E3415A3642D57D77C6C491D6AC1D69

# 2. Agregar el repositorio oficial
echo "deb [signed-by=/usr/share/keyrings/k6-archive-keyring.gpg] https://dl.k6.io/deb stable main" | sudo tee /etc/apt/sources.list.d/k6.list

# 3. Actualizar la lista de paquetes e instalar
sudo apt-get update
sudo apt-get install k6
```

Listing 2: Comandos de instalación de k6 en WSL Ubuntu

4.1.4. Paso 4: Creación de Script de prueba con k6

Se desarrolló el archivo `carga-y-rendimiento.js` (ver Listing 3) configurando los siguientes elementos clave:

- **Carga (Stages):** Simulación de un escalado de hasta 20 usuarios virtuales.
- **Umbrales (Thresholds):** Definición de p(95) $<$ 500ms para latencia y tasa de fallo $<$ 1 %.
- **Función Principal:** Ejecución periódica de peticiones GET y POST con validación de estados y tiempo de espera de 1 segundo entre iteraciones.

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = { ... };

export default function () {
    // Peticion GET
    const resGet = http.get('http://localhost:3333/api/test');
    check(resGet, { 'GET status is 200': (r) => r.status === 200 });

    // Peticion POST
    const payload = JSON.stringify({ user: 'Tester k6' });
    const params = { headers: { 'Content-Type': 'application/json' } };
    const resPost = http.post('http://localhost:3333/api/data', payload
    , params);
    check(resPost, { 'POST status is 201': (r) => r.status === 201 });

    sleep(1);
}
```

Listing 3: Fragmento del script de k6 con GET y POST

Tras la configuración del script, se ejecutó la prueba de carga obteniendo los resultados visualizados en la Figura 4. El análisis detallado de estas métricas se presenta a continuación, basándose en los datos recolectados durante la ejecución.

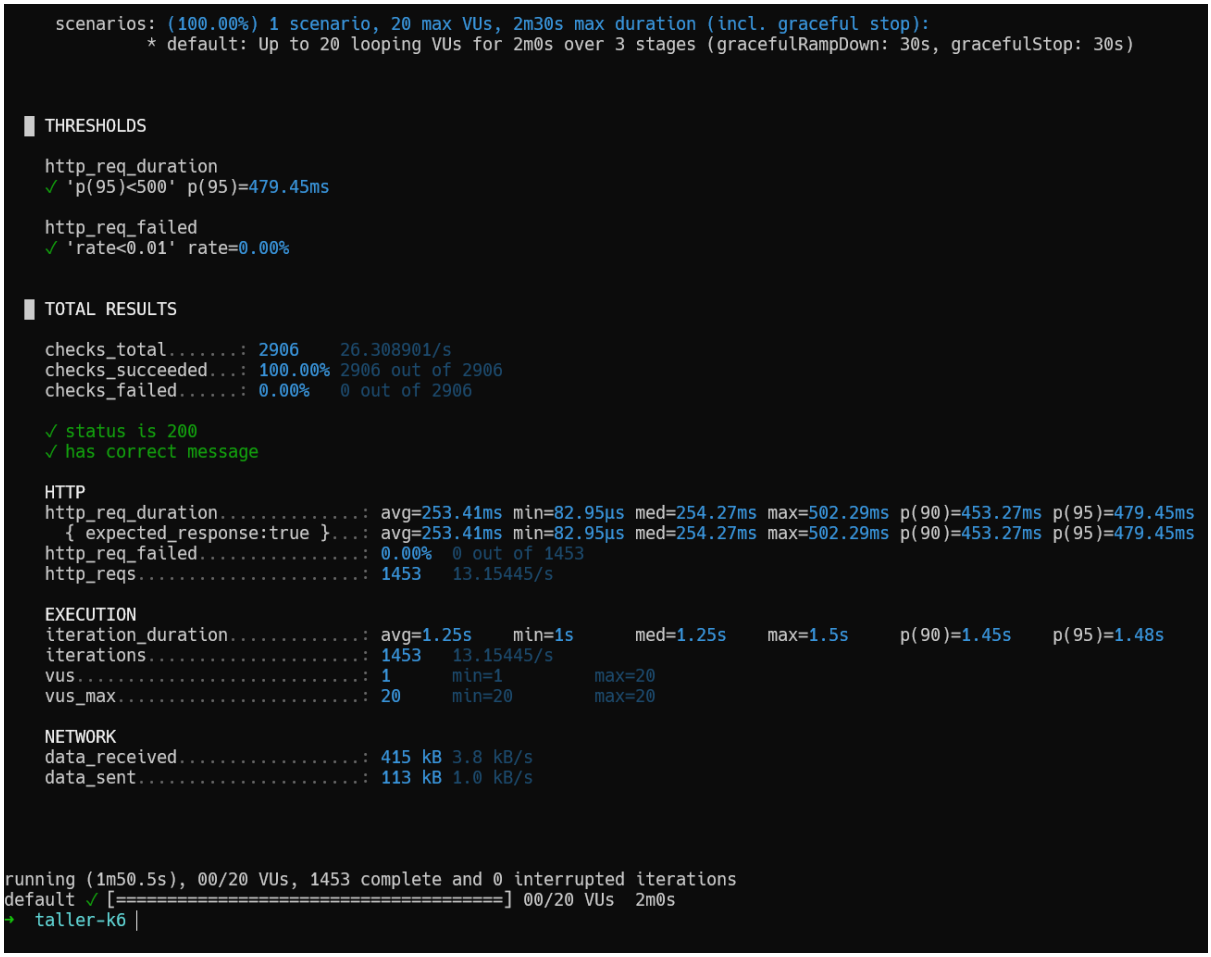

Figura 4: Ejecución exitosa del script de k6 y validación de umbrales.

Análisis de Resultados (k6):

Métrica de Rendimiento	**Valor Obtenido**	**Umbral / Objetivo**	**Estado**
Usuarios Virtuales (VUs)	20	Máximo estipulado	Completado
Iteraciones Totales	1455	N/A	Valido
Tiempo de Respuesta (p95)	473.08 ms	$<$ 500 ms	**PASÓ**
Tasa de Errores (HTTP 4xx/5xx)	0.00 %	$<$ 1 %	**PASÓ**
Verificaciones (Checks) Exitosos	100.00 %	100 %	**PASÓ**
Throughput (Solicitudes/seg)	13.12 reqs/s	N/A	Estable

Cuadro 1: Métricas detalladas de la ejecución de pruebas de carga en el Paso 4.

El análisis de la Tabla 1 demuestra que el sistema mantuvo un comportamiento estable bajo la carga máxima alcanzada (20 VUs). Se realizaron 2910 verificaciones individuales (estado HTTP 200/201 y validación de mensajes), todas con éxito rotundo.

4.2. Parte 2: Realizar pruebas con k6

En esta sección se describirá el proceso de ejecución de las pruebas y la interpretación de los resultados obtenidos.

4.2.1. Paso 1: Ejecución de las pruebas de carga y rendimiento

[Pendiente de completar]

4.2.2. Paso 2: Interpretación de métricas

[Pendiente de completar]

4.2.3. Paso 3: Cambio en la configuración del test

[Pendiente de completar]

4.3. Parte 3: Pruebas de carga y rendimiento a un backend completo

En esta parte se abordará el análisis de rendimiento sobre un sistema backend funcional con mayor complejidad.

4.3.1. Paso 1: Ejecución del backend completo

[Pendiente de completar]

4.3.2. Paso 2: Crear el script para pruebas de carga y rendimiento del backend completo

[Pendiente de completar]

4.3.3. Paso 4: Ejecución del script y revisión de los resultados arrojados

[Pendiente de completar]

5. Resultados

A continuación se presentan los resultados obtenidos de la ejecución de la prueba con 20 usuarios virtuales, los cuales se resumen en la Tabla 2.

5.1. Análisis de Métricas Principales

Métrica	**Umbral**	**Resultado**	**Estado**
http_req_duration (p95)	< 500ms	473.08ms	PASÓ
http_req_failed (rate)	$< 1\,\%$	$0.00\,\%$	PASÓ
Checks de éxito	$100\,\%$	$100.00\,\%$	PASÓ

Cuadro 2: Resumen de umbrales y resultados generales.

5.2. Estadísticas de Latencia

Los tiempos de respuesta registrados muestran una estabilidad adecuada bajo carga:

- **Promedio (avg):** 249.96 ms
- **Mediana (med):** 257.93 ms
- **Máximo (max):** 502.8 ms

El sistema logró procesar un total de **1455 peticiones** con un rendimiento promedio de 13.12 reqs/s, sin registrar fallos durante toda la ejecución.

6. Conclusiones

- El servidor Express demostró ser capaz de manejar 20 usuarios virtuales concurrentes manteniendo un tiempo de respuesta p(95) por debajo de los 500ms, lo cual indica una arquitectura estable para este nivel de carga.
- La herramienta k6 facilitó la identificación de métricas críticas de rendimiento y la validación automática de criterios de calidad mediante el uso de *thresholds*.

Referencias

Bezemer, C. P. et al. (2019). Performance challenges in microservices and proposed solutions. *Communications of the ACM*, 62(10):68–77.

Sahli, A. et al. (2023). Automated load testing for restful apis: A comparative study. In *2023 International Conference on Software Engineering and Knowledge Engineering*.

Vuković, M. et al. (2020). Performance testing of web services: A comparison of jmeter and k6. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1122–1127. IEEE.

7. Anexos

7.1. Código Fuente y Repositorio

Todo el código fuente utilizado en esta práctica, incluyendo el servidor y los scripts de *k6*, se encuentra disponible en el siguiente repositorio de GitHub:

- **Repositorio:** `https://github.com/gamurigm/Pruebas_k6_beta.git`