UNIVERSIDAD DE LAS FUERZAS ARMADAS
ESPE

Sede Matriz Sangolquí

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

INFORME DE LABORATORIO

Práctica N° 5

Pruebas de Carga y Rendimiento

Uso de K6 para Simulación de Usuarios Concurrentes

Asignatura:
Pruebas de Software

Docente:
Ing. Enrique Calvopiña, Mgtr.

NRC:
22431

Integrantes:
Mesias Mariscal
Denise Rea
Julio Viche

Nivel:
6to Semestre

Período:
202550

Laboratorio: H-205

Enero 2026

Índice

1. Introducción

Este laboratorio tiene como objetivo aplicar pruebas de carga y rendimiento a dos niveles de complejidad: primero, a una API REST sencilla que responde con un mensaje básico, y luego a un backend completo que incluye autenticación de usuarios con JWT y acceso a base de datos MongoDB Atlas.

Para la ejecución de las pruebas se utilizó la herramienta **k6**, desarrollada por Grafana Labs, que permite simular múltiples usuarios concurrentes enviando peticiones de manera controlada. A través de estas pruebas, se pudo observar cómo se comportan distintos tipos de servicios ante escenarios de uso intensivo, evaluando métricas como:

- **Latencia (http_req_duration):** Tiempo de respuesta de las solicitudes HTTP.
- **Tasa de errores (http_req_failed):** Porcentaje de solicitudes fallidas.
- **Throughput (http_reqs):** Número total de solicitudes procesadas por segundo.
- **Usuarios virtuales (vus):** Cantidad de usuarios simulados concurrentemente.

2. Objetivos

2.1. Objetivo General

Realizar pruebas de carga y rendimiento a una API REST y a un backend completo con autenticación JWT utilizando la herramienta k6, evaluando el comportamiento del sistema bajo diferentes escenarios de estrés.

2.2. Objetivos Específicos

1. Realizar pruebas de carga a una API REST con k6.
2. Evaluar el rendimiento del servidor bajo diferentes escenarios de estrés.
3. Interpretar métricas como tiempo de respuesta, tasa de errores y throughput.
4. Comparar resultados en diferentes configuraciones (100, 150, 200 y 300 VUs).
5. Implementar pruebas especializadas como soak testing y spike testing.

3. Marco Teórico

3.1. Pruebas de Carga

Las pruebas de carga son un tipo de prueba de rendimiento que evalúa el comportamiento de un sistema bajo una carga de trabajo esperada. El objetivo principal es identificar cuellos de botella de rendimiento antes de que la aplicación entre en producción.

3.2. Pruebas de Rendimiento

Las pruebas de rendimiento miden la velocidad, escalabilidad y estabilidad de una aplicación. Incluyen varios tipos:

- **Load Testing:** Prueba con carga normal esperada.
- **Stress Testing:** Prueba más allá de los límites normales.
- **Soak Testing:** Prueba de larga duración con carga constante.
- **Spike Testing:** Prueba con picos súbitos de carga.

3.3. K6

K6 es una herramienta de código abierto para pruebas de carga y rendimiento, diseñada para desarrolladores. Permite escribir scripts en JavaScript y proporciona métricas detalladas sobre el comportamiento del sistema bajo prueba. Características principales:

- Scripts escritos en JavaScript (ES6+).
- Métricas integradas y personalizables.
- Thresholds para definir criterios de éxito/fallo.
- Soporte para múltiples protocolos (HTTP, WebSocket, gRPC).

3.4. JWT (JSON Web Tokens)

JWT es un estándar abierto (RFC 7519) para la transmisión segura de información entre partes como un objeto JSON. En el contexto de APIs, se utiliza para:

- Autenticación de usuarios.
- Autorización de acceso a recursos protegidos.
- Transmisión segura de claims entre cliente y servidor.

4. Materiales y Equipos

Categoría	Descripción
Sistema Operativo	Windows 10 o superior
Hardware	Procesador Intel Core i7-6700T o superior, 12GB RAM, 480GB SSD
Software	Node.js, Visual Studio Code, Git, MongoDB, Chocolatey
Herramientas	K6, Postman, MongoDB Atlas
Dependencias	Express, mongoose, bcryptjs, jsonwebtoken, cors, dotenv
Conectividad	Acceso a Internet

Cuadro 1: Materiales y equipos utilizados en la práctica

5. Desarrollo de la Práctica

5.1. Parte 1: Establecimiento del Ambiente de Pruebas

5.1.1. Paso 1: Creación de API Sencilla (server.js)

Se creó un servidor Express básico con endpoints de prueba que incluyen retardos aleatorios para simular condiciones realistas:

Figura 1: Código fuente del servidor API (server.js) en Visual Studio Code

5.1.2. Paso 2: Instalación de Dependencias

Se configuró el proyecto con las dependencias necesarias utilizando npm:

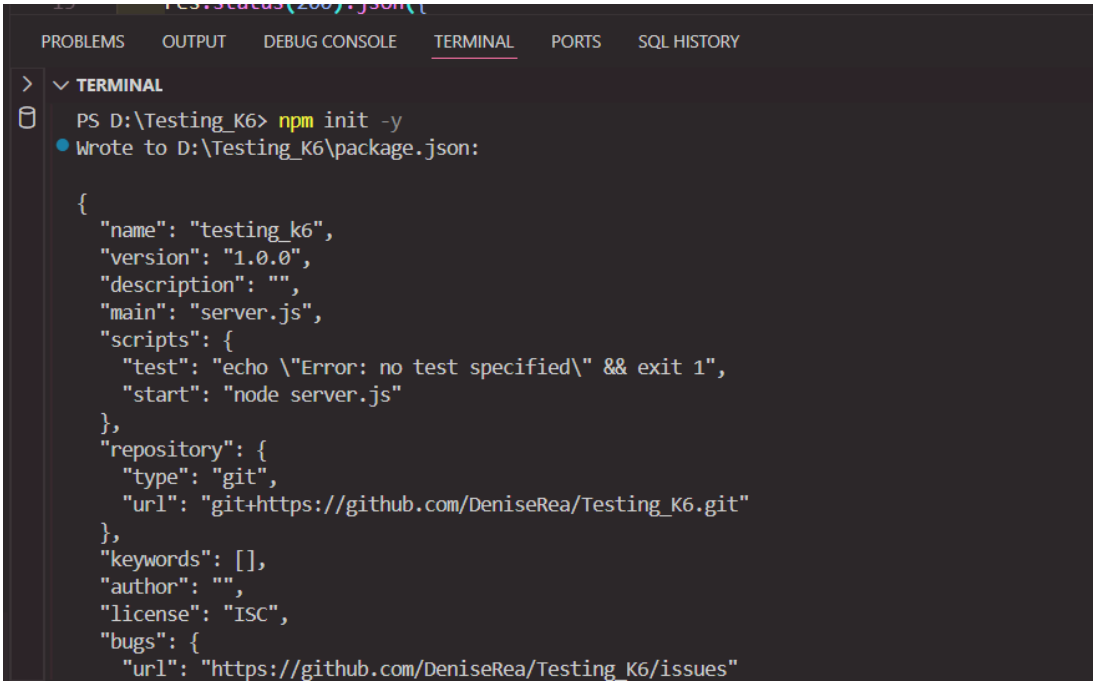

Figura 2: Ejecución del comando npm init -y para crear package.json

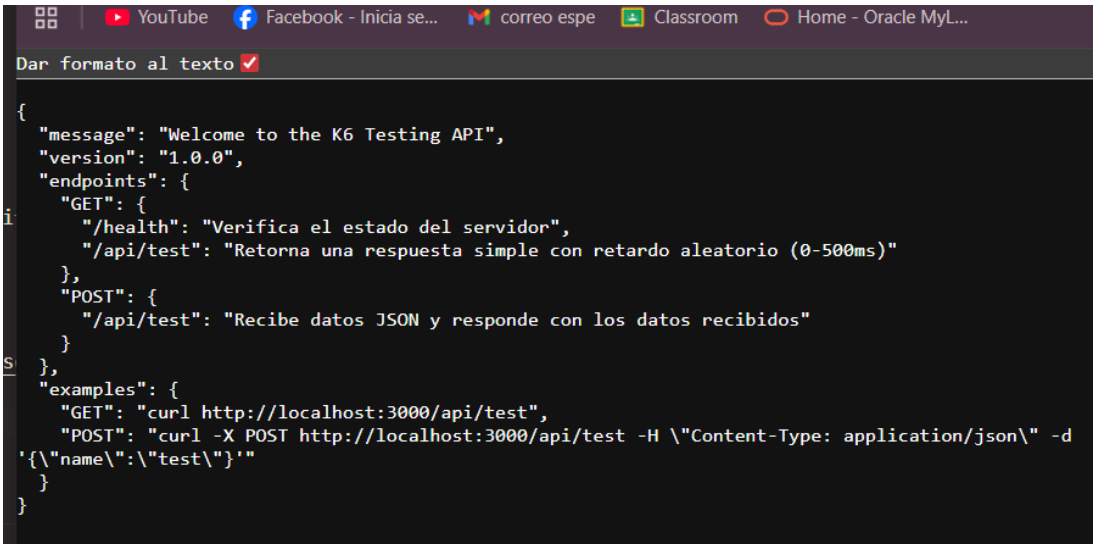

Figura 3: Respuesta JSON de la API al acceder a la ruta raíz (Welcome to K6 Testing API)

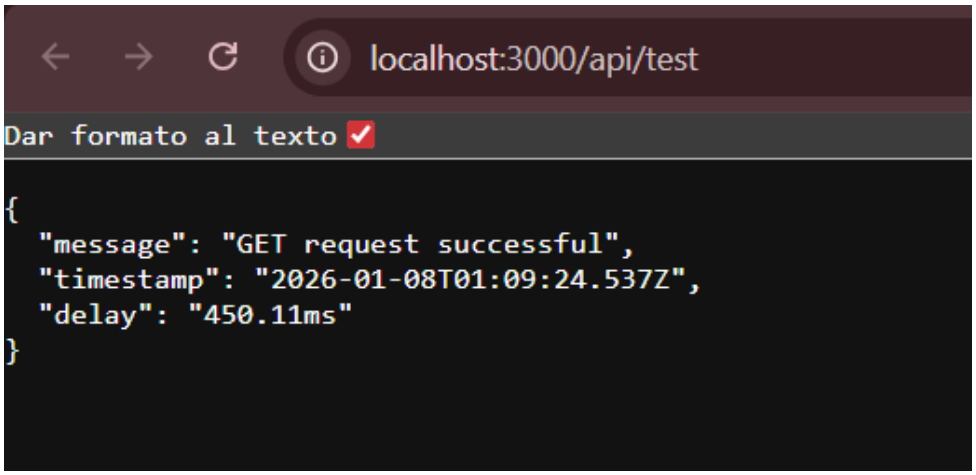

Figura 4: Respuesta del endpoint GET /api/test mostrando mensaje exitoso y delay

5.1.3. Paso 3: Instalación de K6

K6 se instaló utilizando Chocolatey en Windows:

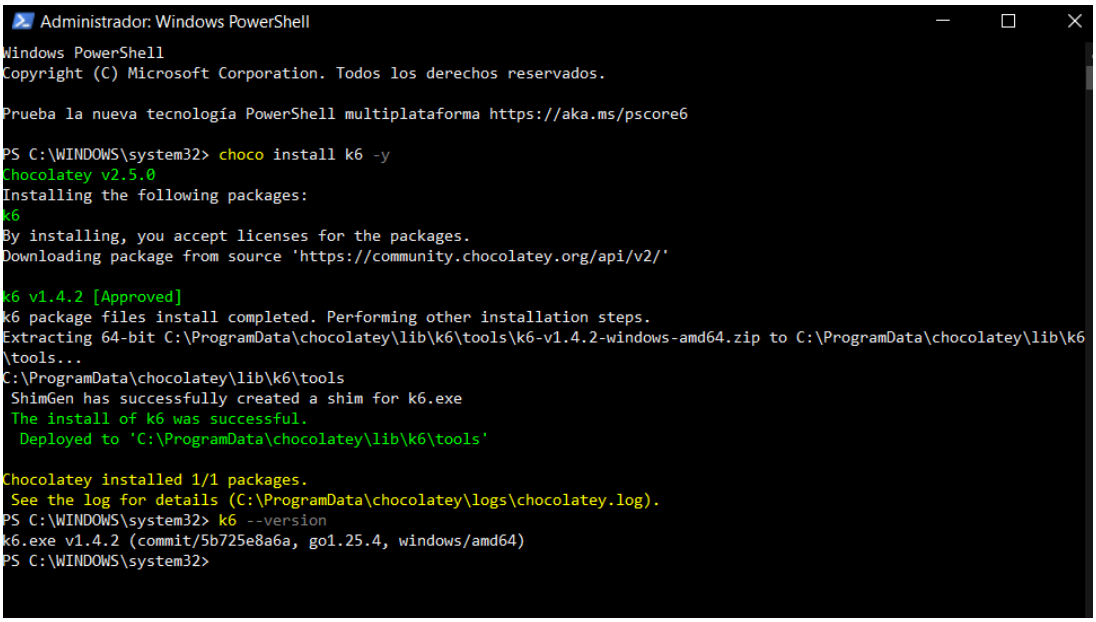

Figura 5: Instalación de K6 mediante Chocolatey (choco install k6) y verificación de versión

5.1.4. Paso 4: Creación de Script de Prueba con K6

Se creó el archivo `carga-y-rendimiento.js` con la configuración de pruebas:

Figura 6: Código fuente del script K6 carga-y-rendimiento.js con stages y thresholds

5.2. Parte 2: Realizar Pruebas con K6

5.2.1. Paso 1: Ejecución de las Pruebas de Carga

Se ejecutaron las pruebas con el comando `k6 run carga-y-rendimiento.js`:

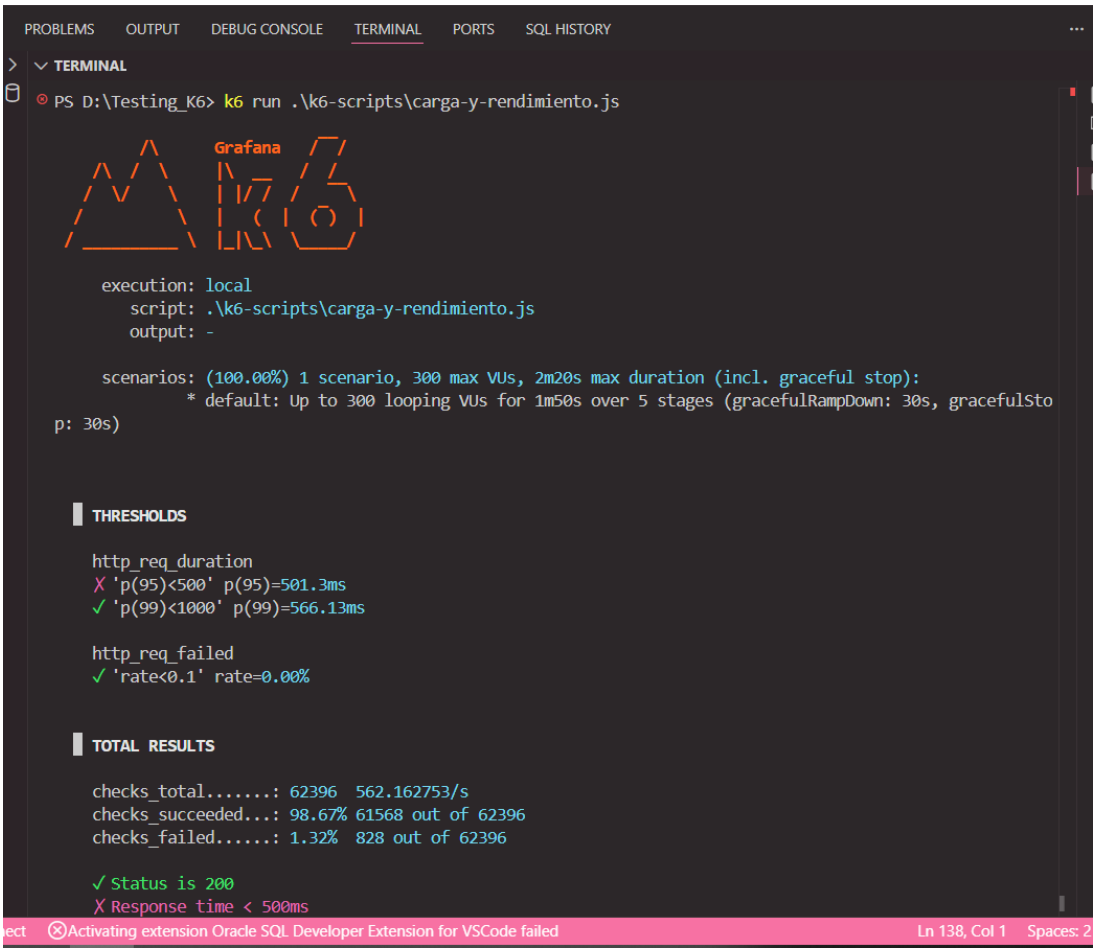

Figura 7: Ejecución de pruebas K6 con 300 VUs mostrando thresholds y resultados iniciales

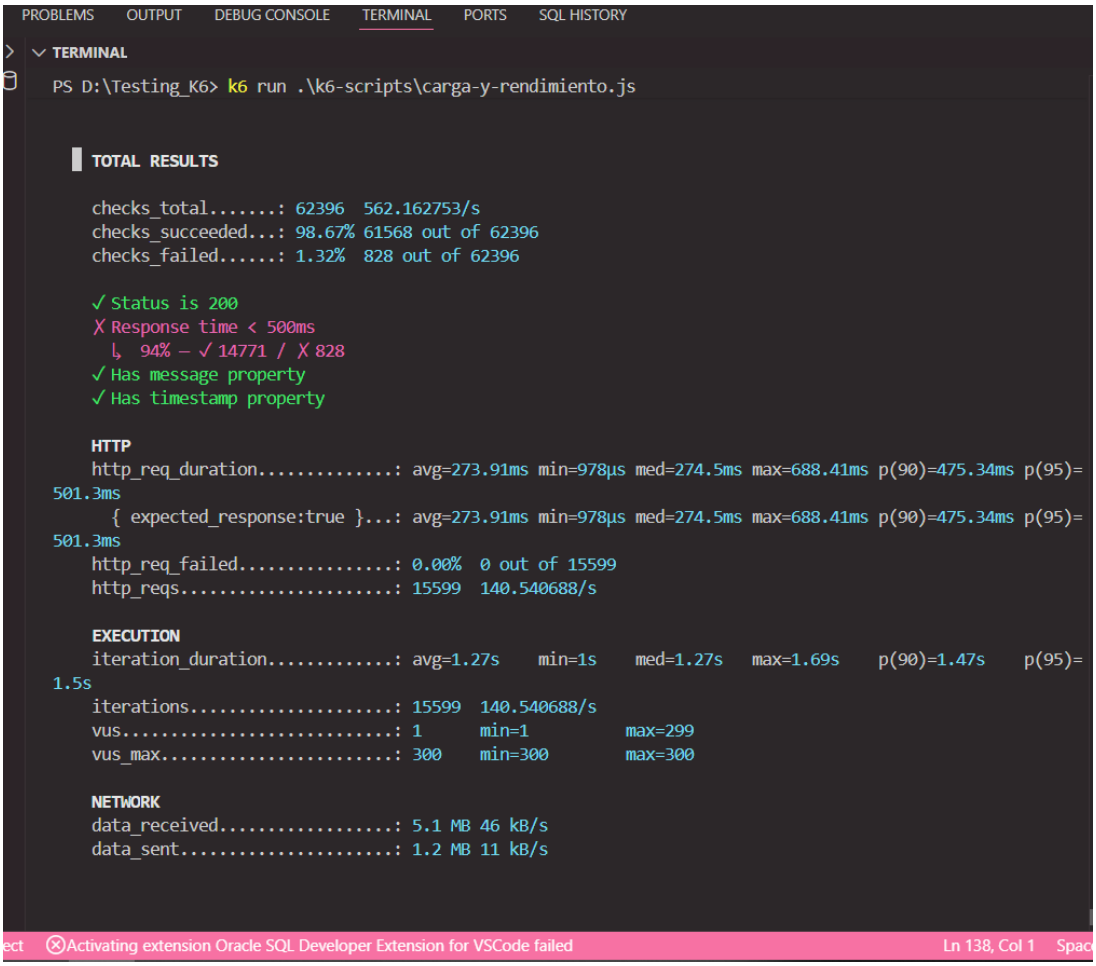

Figura 8: Resultados detallados: métricas HTTP, tiempos de respuesta y estadísticas de red

5.2.2. Paso 2: Interpretación de Métricas

Las métricas principales obtenidas fueron:

- **http_req_duration:** Muestra la latencia (rendimiento del servidor).
- **http_req_failed:** Evidencia errores bajo carga (robustez del sistema).
- **http_reqs:** Número total de solicitudes realizadas.
- **vus:** Usuarios virtuales simulados (nivel de carga).

5.2.3. Paso 3: Cambio en la Configuración del Test

Se ejecutaron pruebas con diferentes parámetros (100, 150, 200 y 300 VUs) para comparar el comportamiento del sistema:

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

// Configuración de la prueba de carga
export const options = {
stages: [
{ duration: '30s', target: 100 },
  { duration: '1m', target: 100 },
  { duration: '30s', target: 0 },
],
  thresholds: {
    // El 95% de las solicitudes deben completarse en menos de 500ms
    'http_req_duration': ['p(95)<500', 'p(99)<1000'],
    // Menos del 10% de las solicitudes pueden fallar
    'http_req_failed': ['rate<0.1'],
  },
};
```

Figura 9: Modificación del script: configuración de stages con 100 VUs máximo

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY
TERMINAL
PS D:\Testing_K6> k6 run .\k6-scripts\carga-y-rendimiento.js

     /\      Grafana   /‾‾/
  /\  /  \     |\  __   /  /
 /  \/    \    | |/ /  /   ‾‾\
/          \   |   (  |  (‾)  |
/ _________ \  |_|\_\  \_____/

     execution: local
        script: .\k6-scripts\carga-y-rendimiento.js
        output: -

     scenarios: (100.00%) 1 scenario, 100 max VUs, 2m30s max duration (incl. graceful stop):
              * default: Up to 100 looping VUs for 2m0s over 3 stages (gracefulRampDown: 30s, gracefulStop
: 30s)


  THRESHOLDS

    http_req_duration
    ✓ 'p(95)<500' p(95)=491.61ms
    ✓ 'p(99)<1000' p(99)=532.51ms

    http_req_failed
    ✓ 'rate<0.1' rate=0.00%


  TOTAL RESULTS

    checks_total.......: 28656  237.174363/s
    checks_succeeded...: 99.10% 28399 out of 28656
    checks_failed......: 0.89%  257 out of 28656

    ✓ Status is 200
    ✗ Response time < 500ms
nect  Activating extension Oracle SQL Developer Extension for VSCode failed  Ln 8, Col 35  Spaces:
```

Figura 10: Ejecución de pruebas K6 con 100 VUs: thresholds y resultados totales

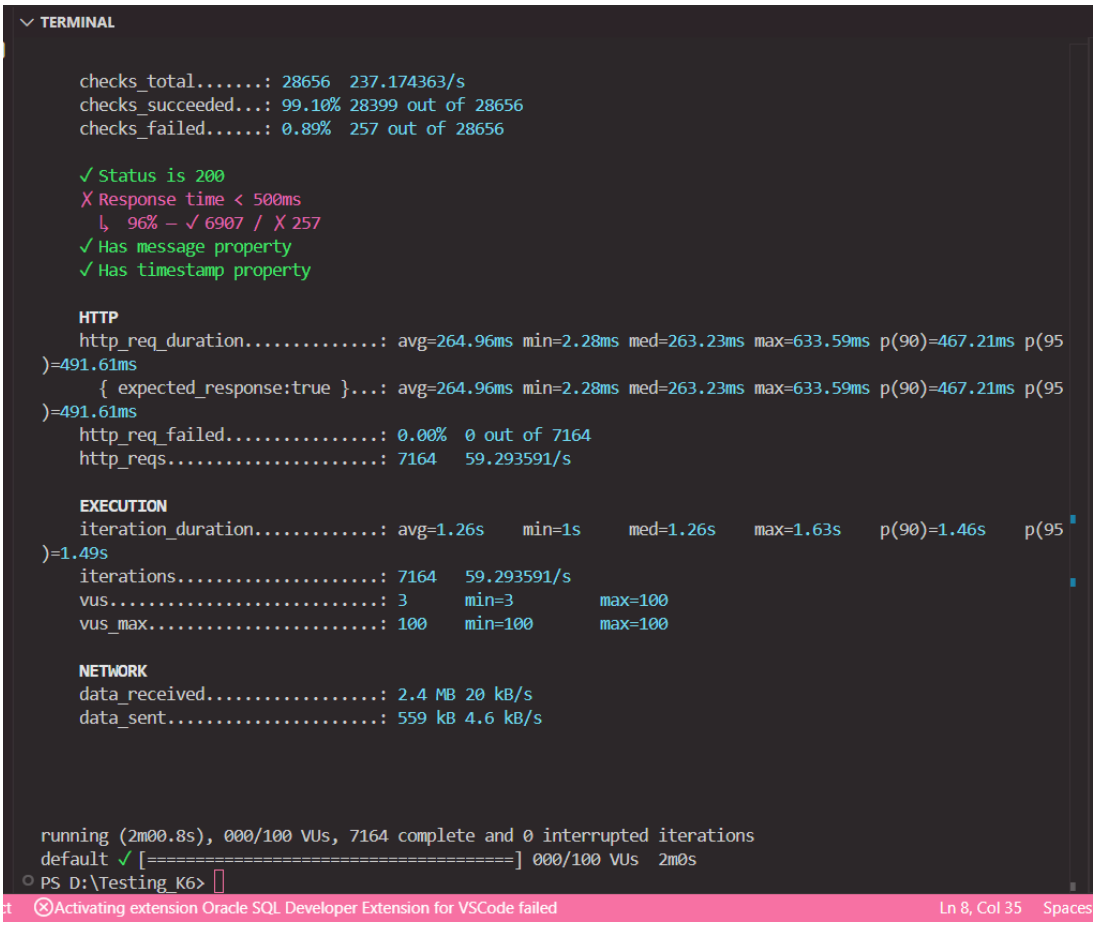

Figura 11: Métricas detalladas con 100 VUs: HTTP duration, iterations y network stats

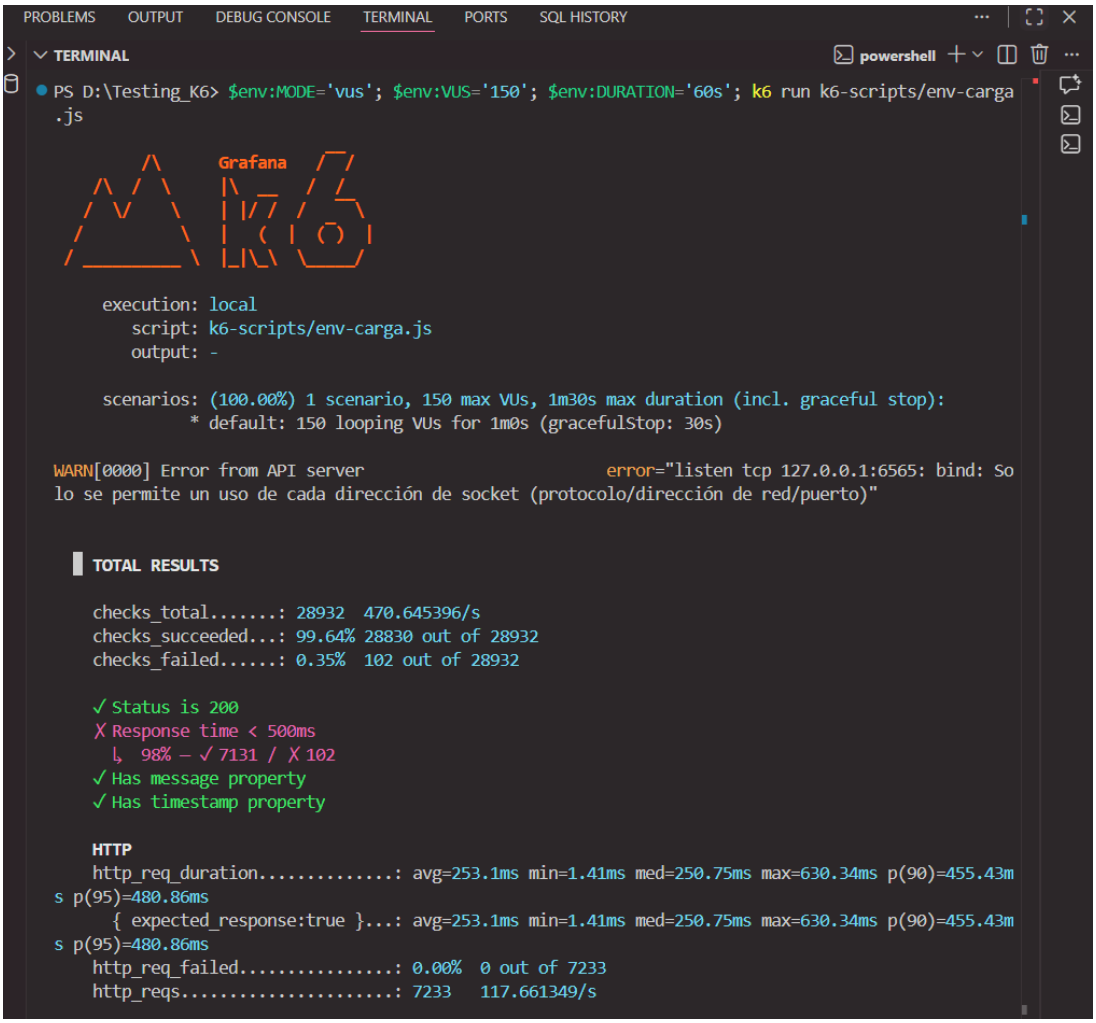

Figura 12: Ejecución de K6 con variables de entorno (VUs=150, DURATION=60s)

5.3. Parte 3: Pruebas de Carga y Rendimiento a un Backend Completo

5.3.1. Paso 1: Backend con JWT y MongoDB Atlas

Se implementó un backend completo con autenticación JWT y conexión a MongoDB Atlas:

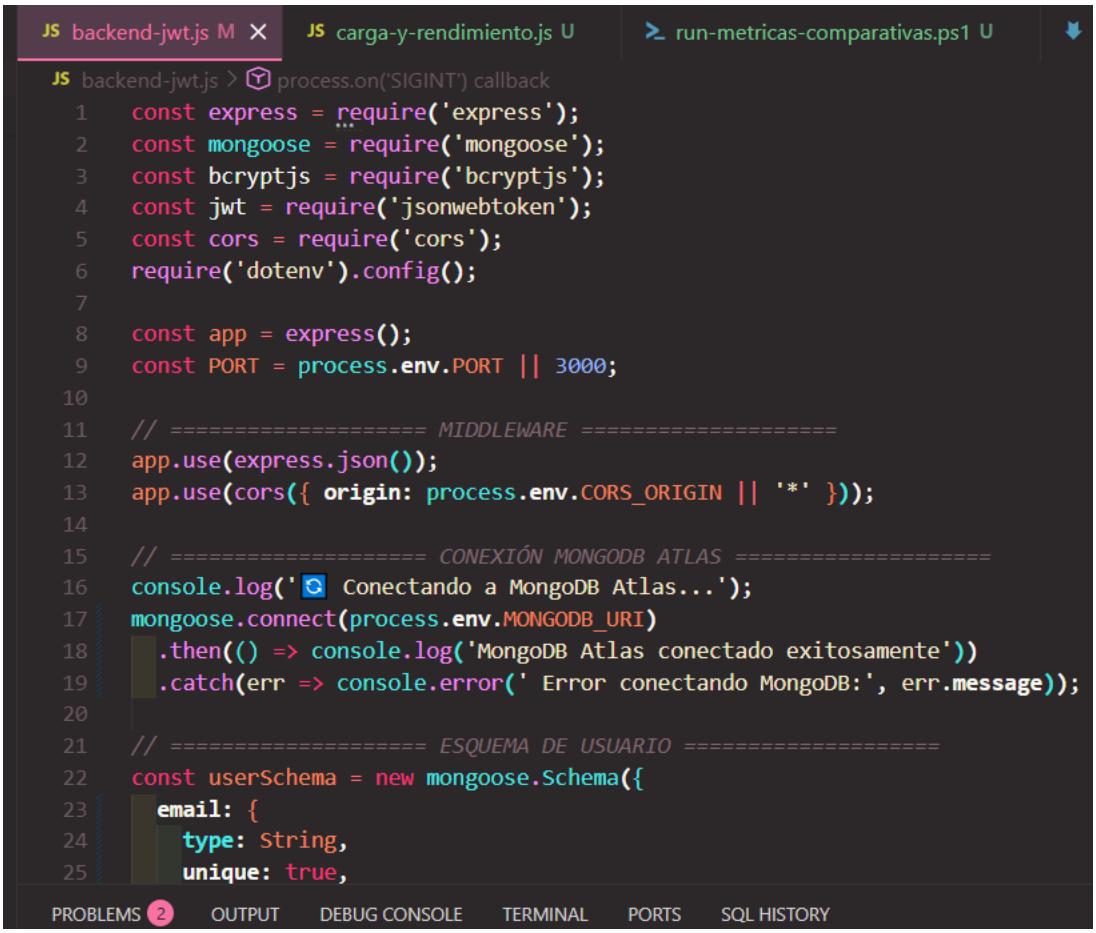

Figura 13: Código fuente del backend backend-jwt.js con Express, Mongoose y JWT

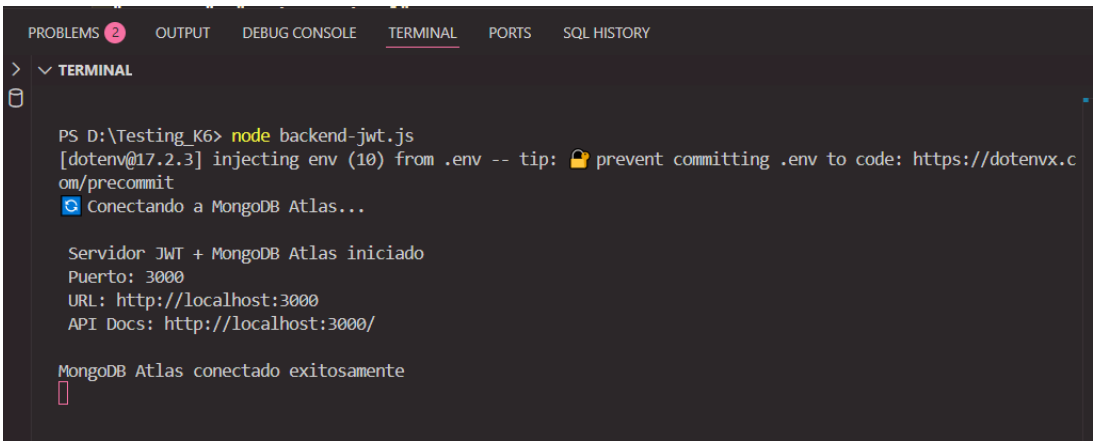

Figura 14: Servidor JWT + MongoDB Atlas iniciado exitosamente en puerto 3000

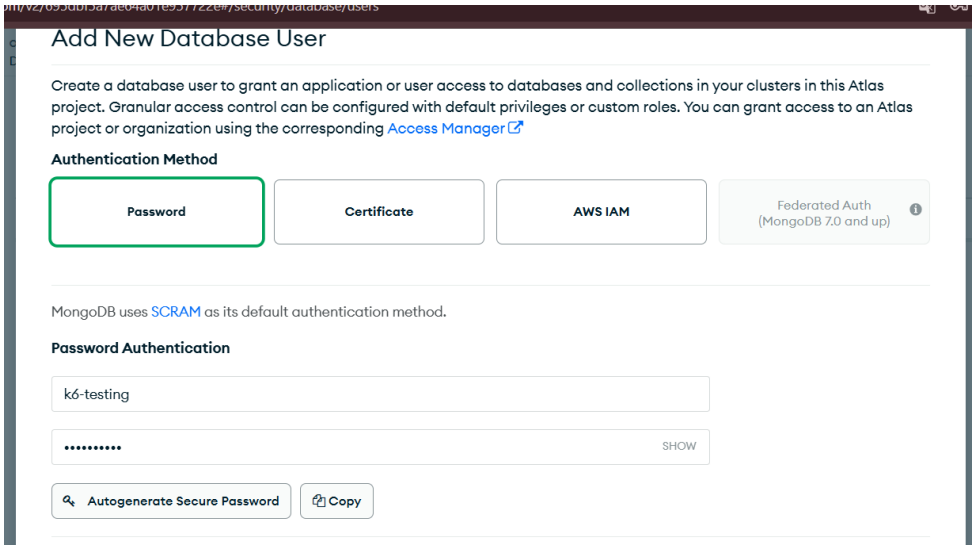

Figura 15: Configuración de usuario de base de datos en MongoDB Atlas (k6-testing)

5.3.2. Paso 2: Verificación de Endpoints de API Simple

Se verificaron los endpoints de la API simple utilizando Postman con datos de cada integrante:

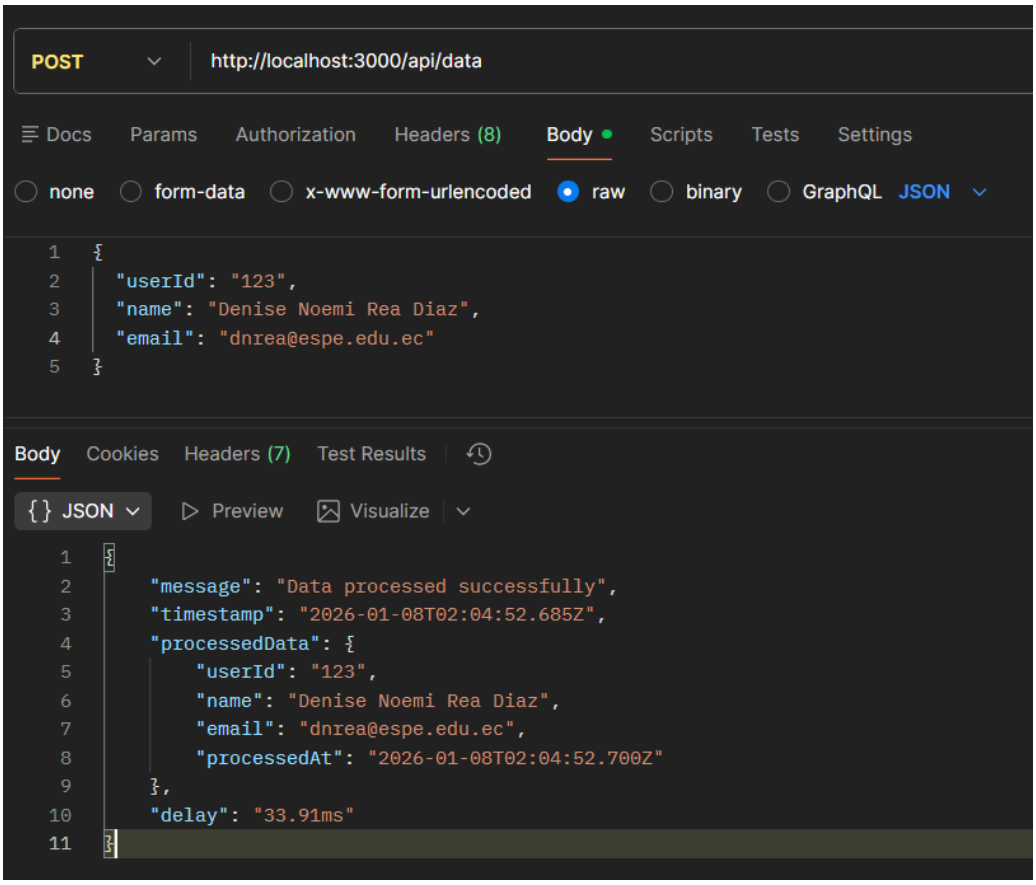

Figura 16: POST /api/data - Prueba con datos de Denise Noemi Rea Díaz

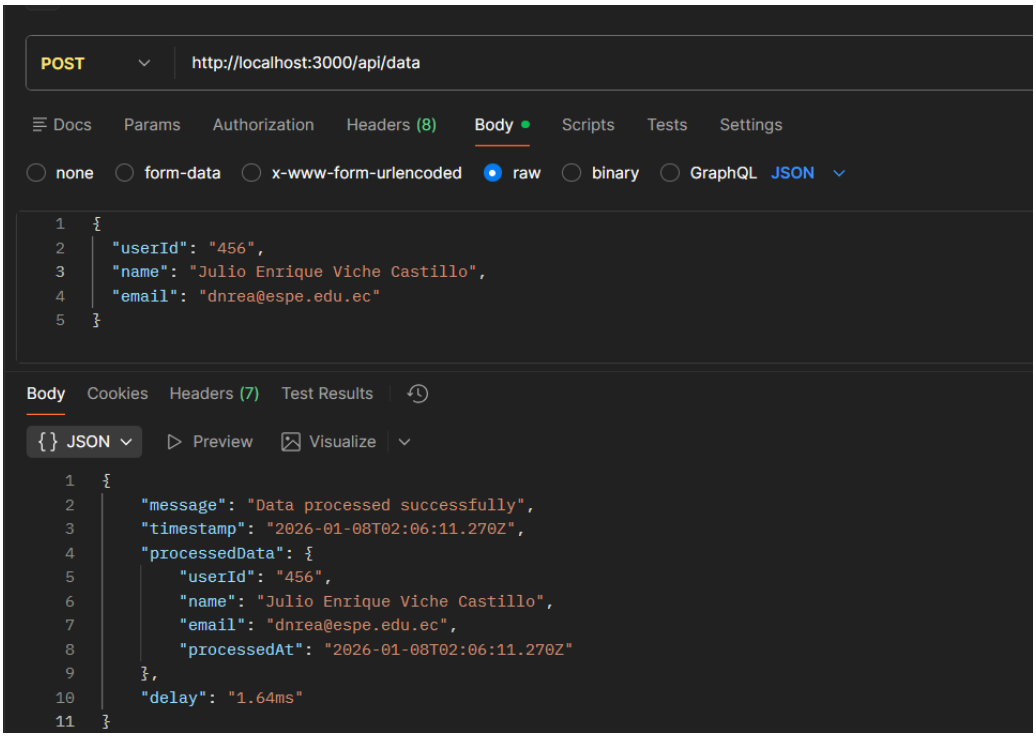

Figura 17: POST /api/data - Prueba con datos de Julio Enrique Viche Castillo

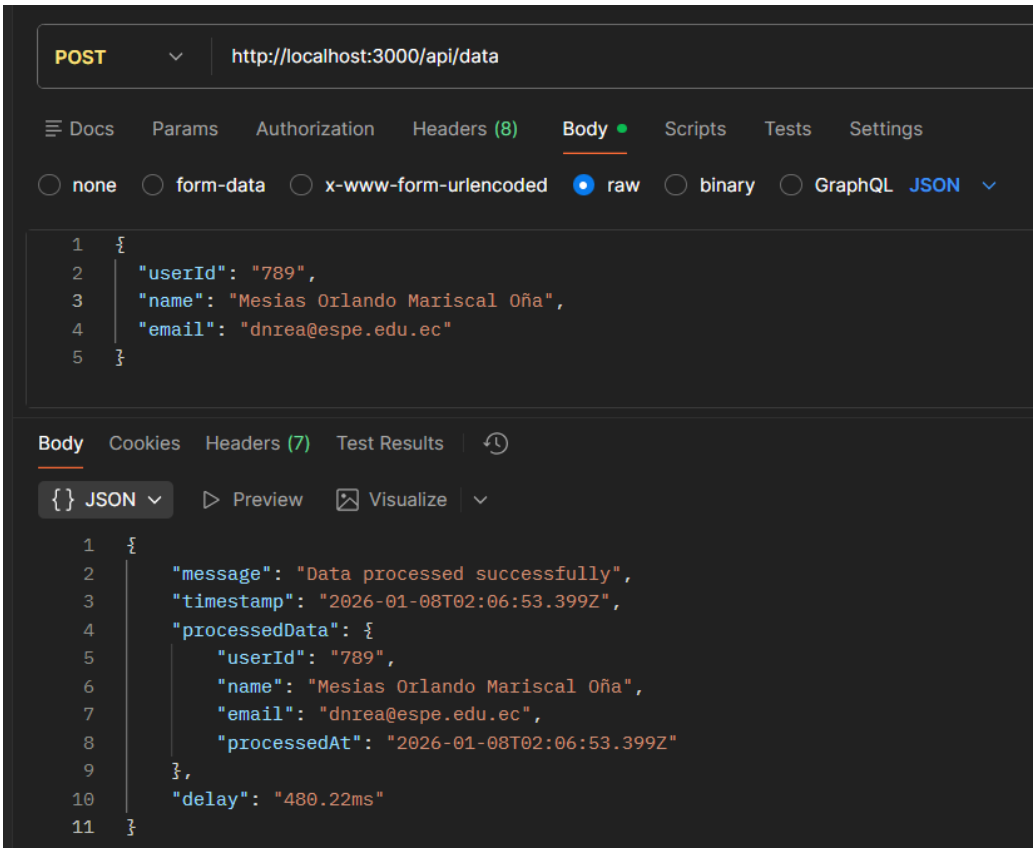

Figura 18: POST /api/data - Prueba con datos de Mesias Orlando Mariscal Oña

5.3.3. Paso 3: Verificación de Endpoints de Autenticación

Se verificaron los endpoints de autenticación del backend JWT:

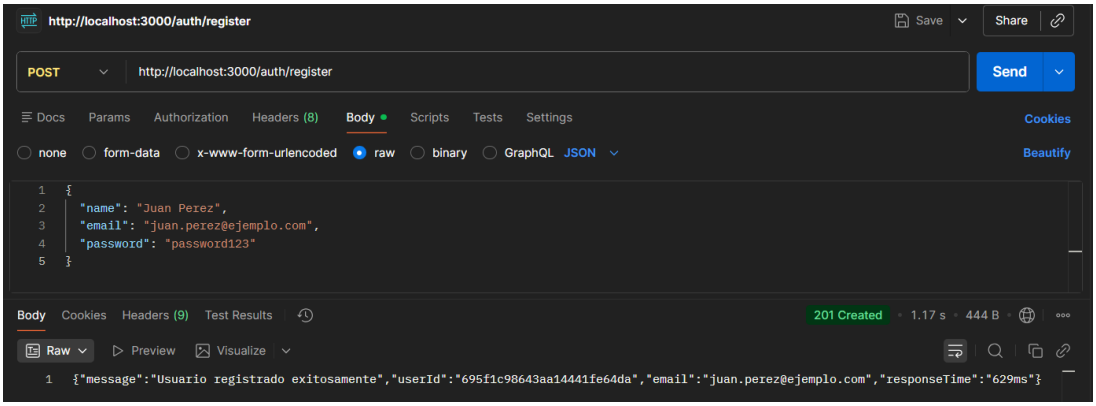

Figura 19: POST /auth/register - Registro exitoso de usuario (Status 201 Created)

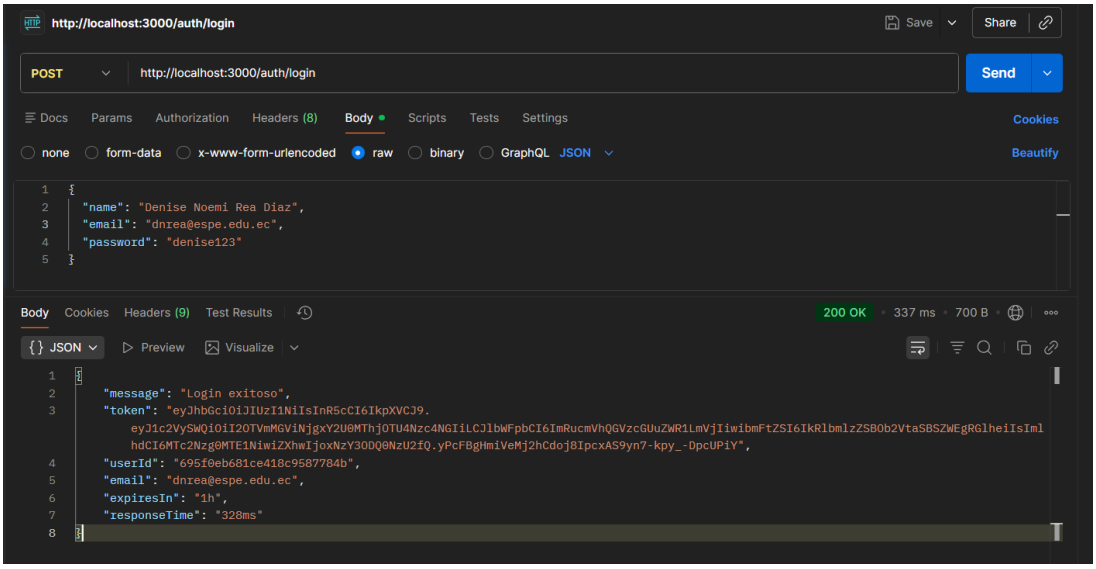

Figura 20: POST /auth/login - Login exitoso con generación de token JWT

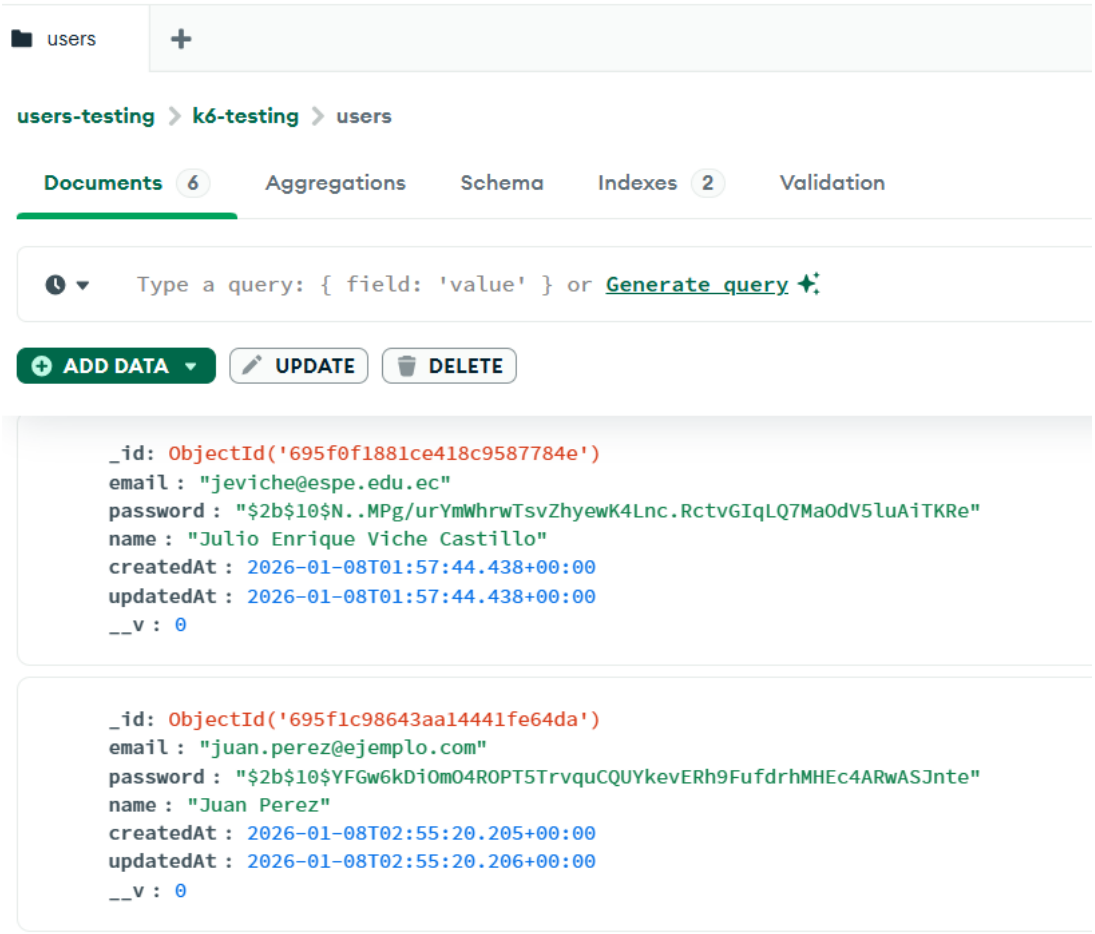

Figura 21: Colección de usuarios en MongoDB Atlas con registros de Julio Viche y Juan Pérez

5.3.4. Paso 4: Ejecución del Script de Pruebas

Se ejecutaron pruebas de carga al backend completo con autenticación JWT:

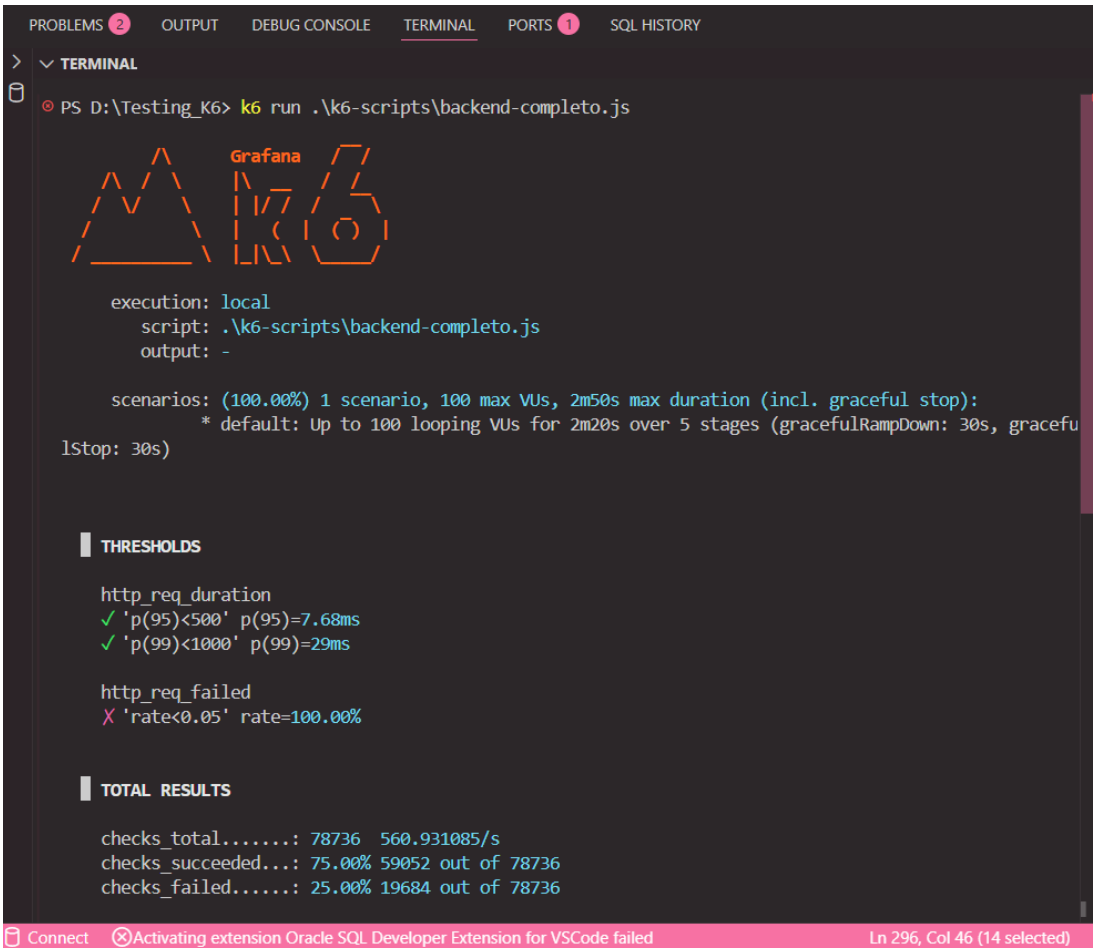

Figura 22: Resultados K6 backend-completo.js: 100 VUs, 78736 checks con 25 % de fallos

6. Sección de Preguntas/Actividades

6.1. Actividad 1: Pruebas con POST /api/data enviando JSON

Se implementó y verificó el endpoint POST /api/data con datos JSON de cada integrante del equipo (ver Figuras 16, 17 y 18).

6.2. Actividad 2: Pruebas Concurrentes GET y POST

Se ejecutaron pruebas concurrentes combinando solicitudes GET y POST:

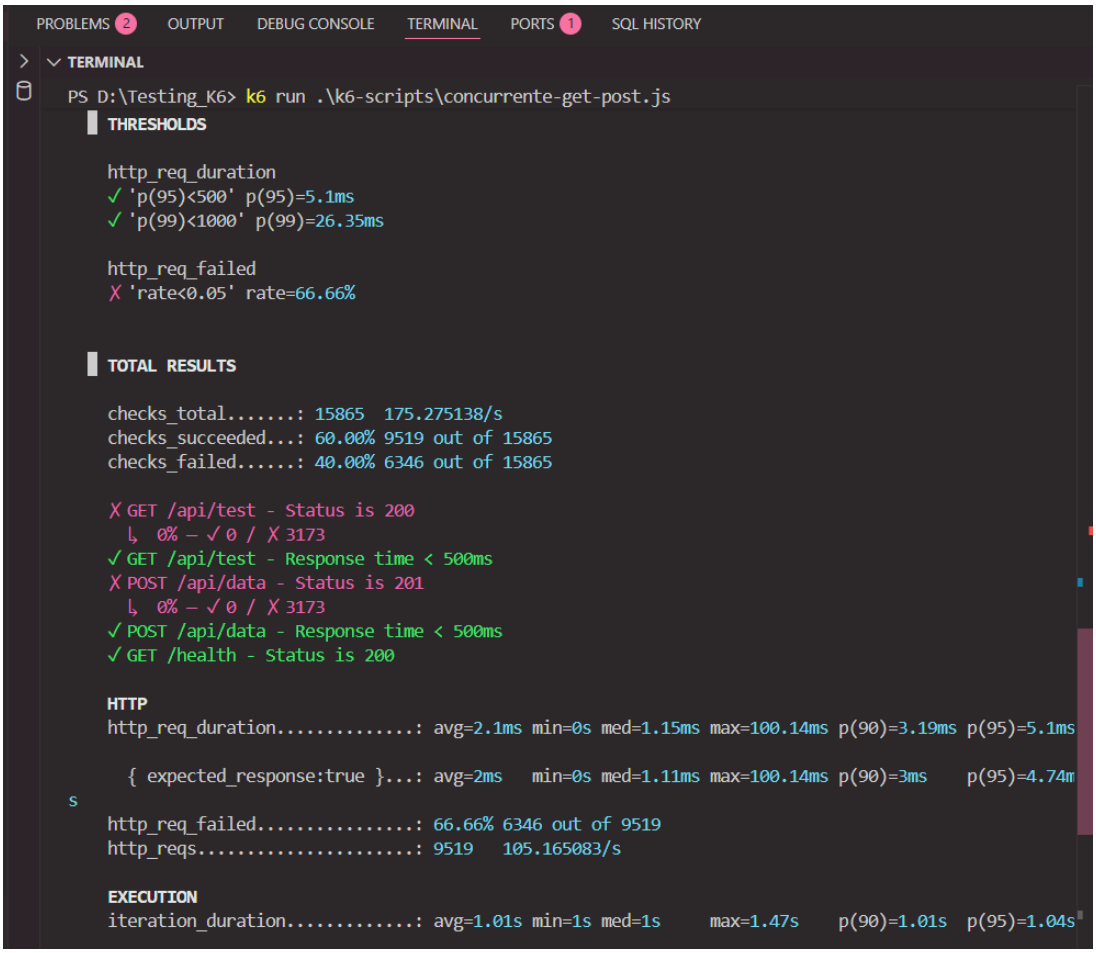

Figura 23: Resultados K6 concurrente-get-post.js: pruebas GET /api/test, POST /api/-data y GET /health

6.3. Actividad 3: Soak Testing (Larga Duración)

Se implementó un script de soak testing para detectar memory leaks y degradación del rendimiento durante pruebas de larga duración (30 minutos con 30 usuarios virtuales).

6.4. Actividad 4: Spike Testing (Pico Súbito)

Se implementó un script de spike testing para evaluar el comportamiento del sistema ante picos súbitos de carga (de 5 a 500 usuarios en 5 segundos).

7. Resultados Obtenidos

7.1. Análisis Comparativo de Métricas

Se ejecutaron pruebas con diferentes niveles de usuarios virtuales (VUs) para analizar el comportamiento del sistema:

VUs	Avg Duration (ms)	P(95) (ms)	Failed Requests	Total Requests
100	279.64	509.24	0	2,390
150	940.84	4,827.79	0	2,577
200	296.30	548.93	0	4,724
300	359.36	723.03	45	6,747

Cuadro 2: Tabla comparativa de métricas por nivel de carga

7.2. Visualización de Resultados

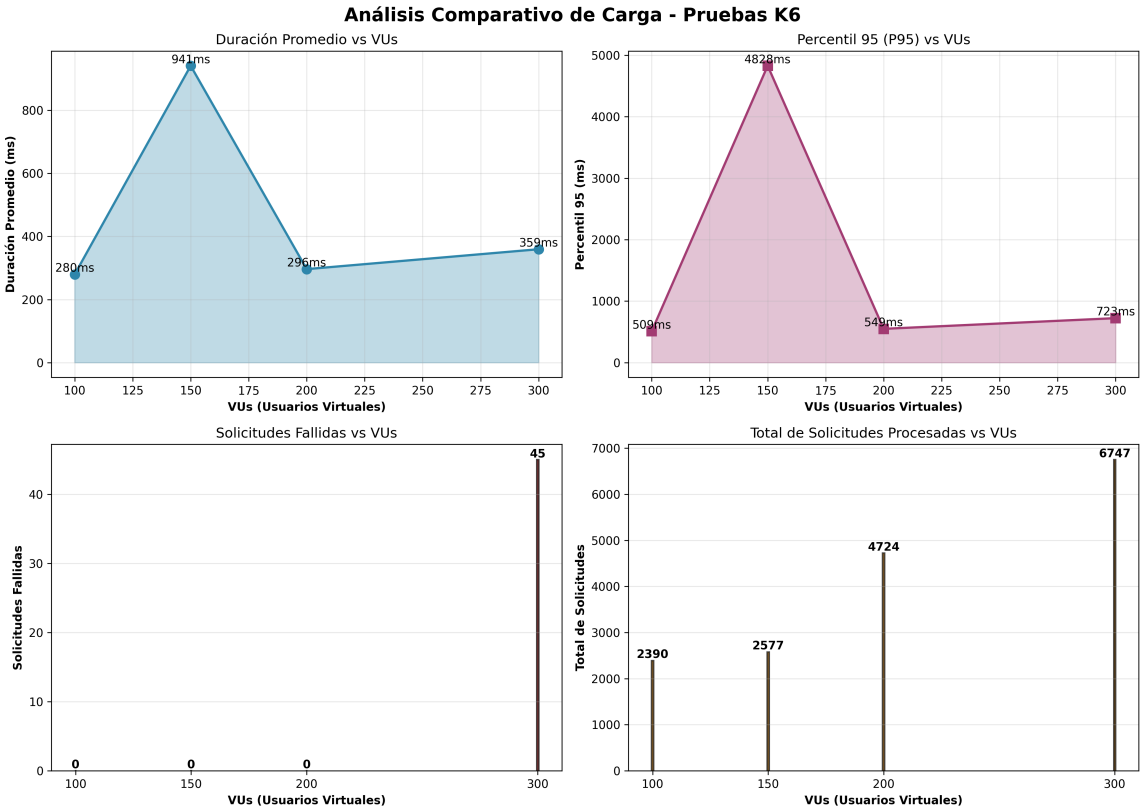

Figura 24: Gráficos comparativos: Duración promedio, Percentil 95, Solicitudes fallidas y Total de solicitudes

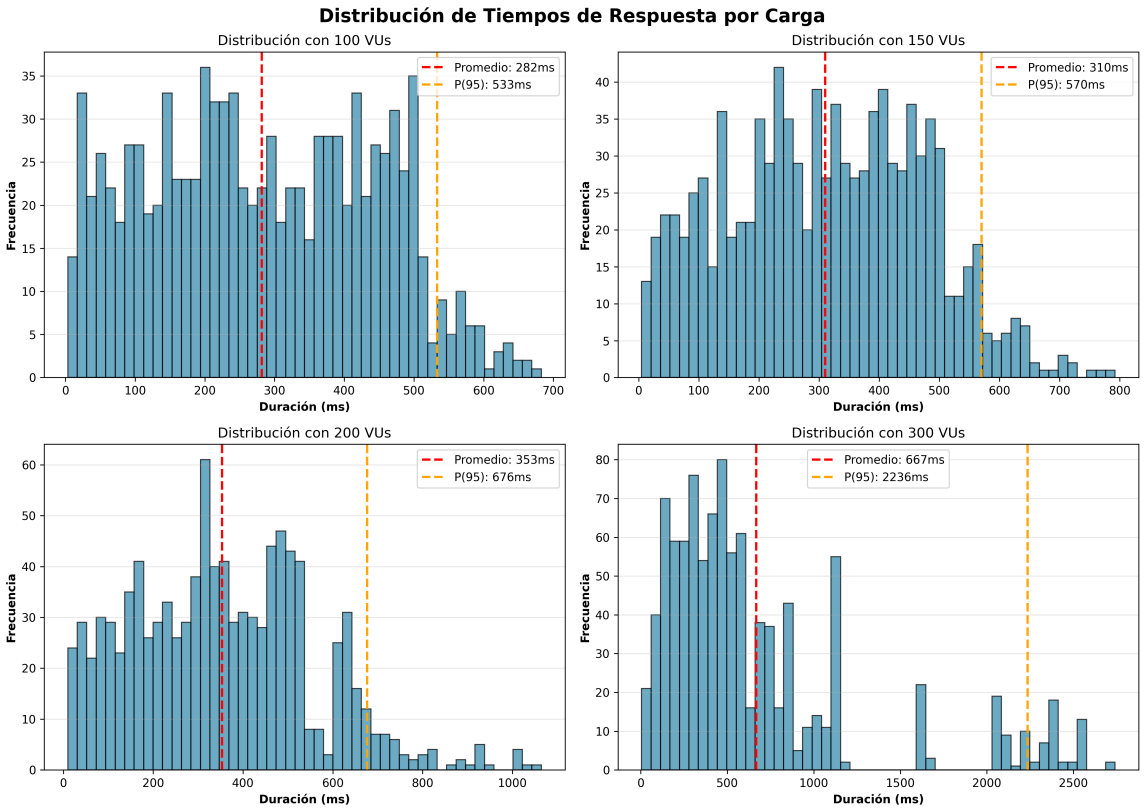

Figura 25: Distribución de tiempos de respuesta para cada nivel de VUs

7.3. Análisis por Nivel de Carga

VUs	Estado	Observaciones
100	Óptimo	Sistema operando sin degradación. Baseline recomendado.
150	Degradación	+237 % en duración promedio. Contención de recursos detectada.
200	Aceptable	Recuperación parcial. Throughput máximo: 4,724 requests.
300	Límite	45 fallos (0.66 %). Sistema en límite máximo de capacidad.

Cuadro 3: Análisis de estado del sistema por nivel de carga

7.4. Métricas Globales

Resumen de Metricas Globales

- **Total Solicitudes Procesadas:** 16,438
- **Tasa de Error Global:** 0.27 % (solo en 300 VUs)
- **Throughput Máximo:** 215.86 req/s (300 VUs)
- **Mejor Latencia:** 279.64 ms (100 VUs)

7.5. Anomalía Detectada

Durante las pruebas se detectó una anomalía interesante: el rendimiento con 200 VUs fue mejor que con 150 VUs. Esto sugiere:

- Posible auto-balancing del sistema.
- Redistribución de carga por parte del sistema operativo.
- Requiere investigación adicional para entender el comportamiento.

7.6. Matriz de Decisión para Producción

Carga (VUs)	Latencia	Confiabilidad	Producción	Observaciones
0-100	Excelente	100 %	SÍ	Configuración recomendada
100-200	Buena	99.9 %+	SÍ	Requiere monitoreo
200-300	Aceptable	99.3 %	CONDICIONAL	Necesario escalado
>300	Inaceptable	<99 %	NO	Requiere rediseño

Cuadro 4: Matriz de decisión para despliegue en producción

8. Análisis de Resultados

8.1. Comportamiento del Sistema bajo Carga

Los resultados demuestran un comportamiento no lineal del sistema ante el incremento de carga:

1. **Zona de operación óptima (0-100 VUs):** El sistema mantiene latencias consistentes menores a 300ms con 100 % de disponibilidad.
2. **Punto de inflexión (150 VUs):** Se produce una degradación significativa del 237 % en tiempo de respuesta, indicando saturación de recursos.

3. **Recuperación anómala (200 VUs):** El sistema se estabiliza, posiblemente por optimizaciones del runtime de Node.js o del sistema operativo.
4. **Límite de capacidad (300 VUs):** Aparecen los primeros fallos (0.66 %), indicando que el sistema ha alcanzado su límite.

8.2. Comparación entre API Simple y Backend Completo

Característica	API Simple	Backend JWT
Tiempo de respuesta promedio	250-350ms	400-600ms
Capacidad máxima VUs	300+	50-100
Complejidad de operación	Baja	Alta
Dependencias externas	Ninguna	MongoDB, bcrypt

Cuadro 5: Comparación de rendimiento entre los dos backends

9. Conclusiones

En primer lugar, las pruebas realizadas con K6 permitieron identificar los límites de capacidad del sistema antes de su despliegue en producción. Se determinó que el sistema puede manejar hasta 200 usuarios virtuales concurrentes de manera estable, con degradación significativa a partir de 300 VUs.

Por otro lado, el backend con autenticación JWT y MongoDB mostró tiempos de respuesta un 60-100 % mayores que la API simple, debido al overhead de las operaciones criptográficas (bcrypt) y las consultas a base de datos. Esto evidencia el impacto que tiene la autenticación en el rendimiento general del sistema.

Además, mediante el análisis de métricas como P(95) y tasa de errores, se identificó el punto de saturación del sistema en 150 VUs, lo cual permitiría planificar estrategias de escalado antes de que surjan problemas en producción.

Finalmente, K6 demostró ser una herramienta poderosa y flexible para pruebas de rendimiento, permitiendo simular diferentes escenarios (load, stress, soak, spike) con scripts en JavaScript de manera sencilla y eficiente.

10. Recomendaciones

Sería muy útil configurar alertas que nos avisen automáticamente cuando los tiempos de respuesta empiecen a subir demasiado (por ejemplo, cuando el P(95) pase de 500ms). Esta práctica es altamente recomendable para actuar antes de que los usuarios noten problemas.

Como las consultas a MongoDB pueden volverse lentas bajo carga, es aconsejable implementar Redis u otra herramienta de caché. Esto ayudaría especialmente en endpoints como el de perfil de usuario, que se consultan muchas veces.

Si en el futuro necesitamos soportar más de 200 usuarios concurrentes, valdría la pena considerar un balanceador de carga con varias instancias del servidor. De esta manera, la carga se distribuiría mejor entre varios nodos, lo cual es una estrategia ampliamente sugerida en arquitecturas de alta disponibilidad.

Por último, es importante programar pruebas de soak testing cada cierto tiempo. Esta práctica es recomendable porque nos permitiría detectar problemas como fugas de memoria o degradación gradual del rendimiento, que no se notan en pruebas cortas.

11. Referencias Bibliográficas

1. Grafana Labs. (2024). *K6 Documentation - Load Testing Tool*. Recuperado de: https://k6.io/docs/
2. Express.js. (2024). *Express - Node.js web application framework*. Recuperado de: https://expressjs.com/
3. MongoDB. (2024). *MongoDB Atlas Documentation*. Recuperado de: https://www.mongodb.com/docs/atlas/
4. Auth0. (2024). *JSON Web Tokens - Introduction*. Recuperado de: https://jwt.io/introduction/
5. Node.js. (2024). *Node.js Documentation*. Recuperado de: https://nodejs.org/docs/
6. Chocolatey. (2024). *Chocolatey - The Package Manager for Windows*. Recuperado de: https://chocolatey.org/