

Proyecto de Redes

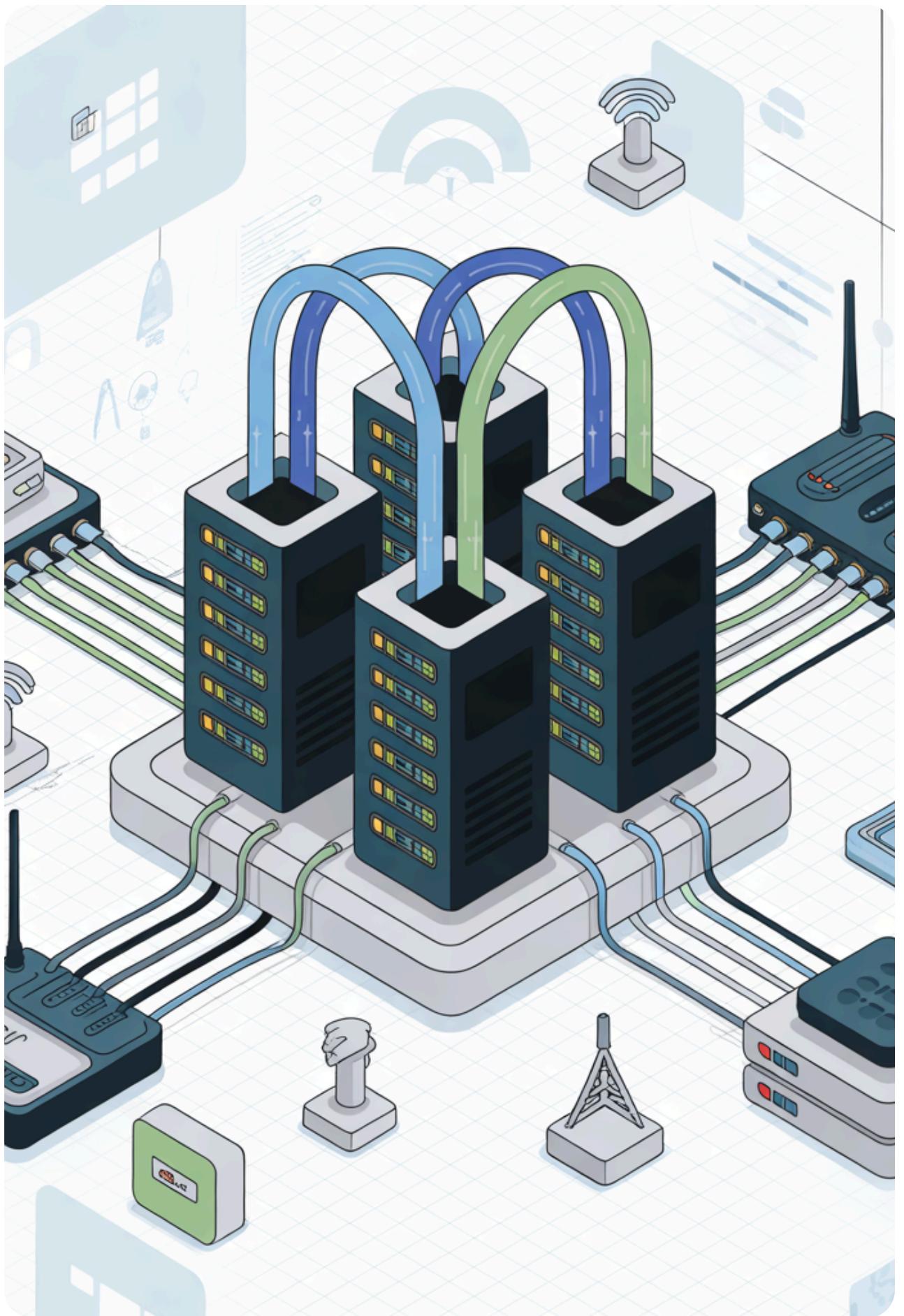
Implementación y monitoreo de un
servidor en entorno virtual

Eliezer Cario (18-10605)

Angel Rodriguez (15-11669)

Kevin Briceño (15-11661)

Miguel Salomon (19-10274)



Problema y motivación

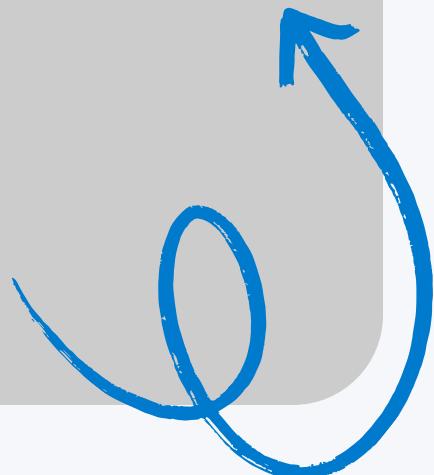
Problema: Las aplicaciones web deben responder a solicitudes concurrentes desde clientes móviles y web; la aleatoriedad del tráfico complica la planificación de capacidad.

Motivación: medir, predecir y diseñar respuestas automáticas para mantener SLA (latencia, disponibilidad).

La meta es que nos podamos adaptar a una demora de respuesta en un servidor web.

Objetivo general:

- Construir un servidor reproducible (VM) y generar carga controlada.
- Recolectar métricas (CPU, memoria, I/O, red, latencia).
- Identificar umbrales y proponer respuestas ante picos.



¿Cómo fue creado el servidor?

Virtualización



Máquina virtual creada con VirtualBox sobre un equipo físico con Windows

Ubuntu



Entorno estable y compatible con servicios de red

Recursos de Hardware VM



8 núcleos de CPU

8 GB de RAM

50 GB de almacenamiento

Modo de red: Bridge Adapter

¿Cómo fue creado el servidor?

Red privada



tailscale

Crear una red privada segura (simulación de LAN online)

SSH



A screenshot of a terminal window showing SSH log output. The text includes error messages like "authentication failure" and "Failed password for root from 192.168.1.3". The word "strongdm" is visible at the bottom left of the terminal window.

```
egrep "Failed|failure" /var/log/auth.log
:17 adc1 sshd[41458]: pam_unix(sshd:auth): authentication failure; logname=
er= rhost=192.168.1.3 user=root
:20 adc1 sshd[41458]: Failed password for root from 192.168.1.3 port 37362 s
:23 adc1 sshd[41458]: Failed password for root from 192.168.1.3 port 37362 s
:28 adc1 sshd[41458]: Failed password for root from 192.168.1.3 port 37362 s
:28 adc1 sshd[41458]: PAM 2 more authentication failures; logname= uid=0 eui
=192.168.1.3 user=root
:11 adc1 sshd[41469]: pam_unix(sshd:auth): authentication failure; logname=
er= rhost=192.168.1.3 user=root
:20 adc1 sshd[41469]: Failed password for root from 192.168.1.3 port 37362 s
:23 adc1 sshd[41469]: Failed password for root from 192.168.1.3 port 37362 s
:28 adc1 sshd[41469]: Failed password for root from 192.168.1.3 port 37362 s
:28 adc1 sshd[41469]: PAM 2 more authentication failures; logname= uid=0 eui
=192.168.1.3 user=root
:11 adc1 sshd[41491]: pam_unix(sshd:auth): authentication failure; logname=
er= rhost=192.168.1.245 user=root
:20 adc1 sshd[41491]: Failed password for root from 192.168.1.245 port 52882 s
:23 adc1 sshd[41491]: Failed password for root from 192.168.1.245 port 52882 s
:28 adc1 sshd[41491]: Failed password for root from 192.168.1.245 port 52882 s
:28 adc1 sshd[41491]: PAM 2 more authentication failures; logname= uid=0 eui
=192.168.1.245 user=root
:42 adc1 sshd[41506]: pam_unix(sshd:auth): authentication failure; logname=
er= rhost=192.168.1.245 user=admin
:42 adc1 sshd[41506]: pam_unix(sshd:auth): authentication failure; logname=
er= rhost=192.168.1.245 user=admin
:42 adc1 sshd[41506]: pam_winbind(sshd:auth): request wbcLogonUser failed: w
berror: PAM_AUTH_ERR (7), NTSTATUS: NT_STATUS_LOGON_FAILURE, Error message was
:45 adc1 sshd[41506]: Failed password for admin from 192.168.1.245 port 5288
```

Conexión remota y segura al servidor

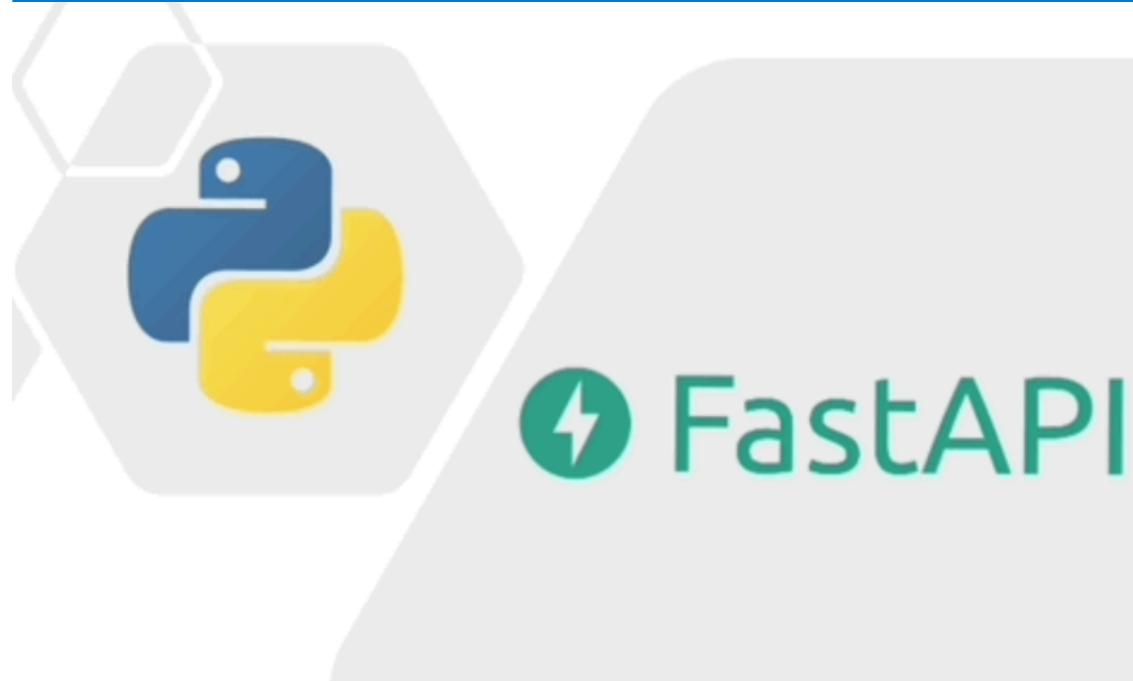
Lenguaje principal y
Herramientas de análisis



Automatización del monitoreo y recolección de
métricas del sistema y red, los resultados en .csv
con timestamps UTC

Propuesta

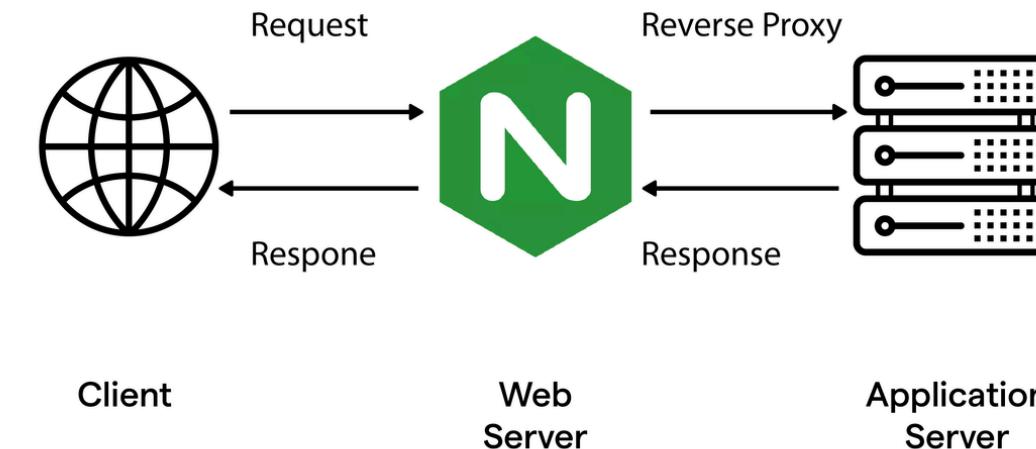
Capa de aplicación



Aplicación de FastAPI actúa como generador controlado de carga de CPU para las pruebas. Exponiendo un endpoint determinista que realiza operaciones matemáticas $O(N)$ para ajustar el esfuerzo de cómputo de manera lineal.

Servidor Web

NGINX



Nginx opera como frontera pública que organiza y distribuye el tráfico hacia la aplicación, aporta medidas básicas de seguridad y centraliza la observabilidad.

Visualización de datos

Visualización en tiempo real

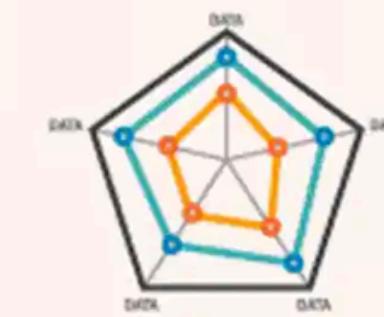
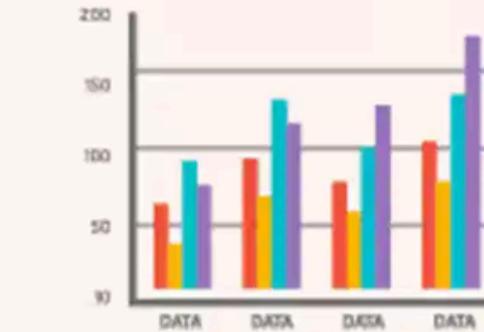


Flask

Aplicación de Flask para un dashboard de visualización de gráficos interactivos en tiempo real. El dashboard Flask lee los CSVs y transmite los datos via Server-Sent Events (SSE) y se renderiza usando Chart.js

Visualización post-mortem

matplotlib



Convertiremos los CSV recolectados por los scripts en hallazgos mediante un script que leerá las series de tiempo de los datos recolectados y generará gráficas claras por métrica y un resumen estadístico

Scripts implementados

Desarrollados en Bash para monitorear rendimiento en tiempo real:

- monitor_cpu.sh: Registra uso de CPU, memoria y carga promedio
- monitor_io.sh: Mide operaciones de entrada/salida en disco
- monitor_net.sh: Captura tráfico de red (bytes recibidos y enviados)
- monitor_latency.sh: Mide latencia total hacia un endpoint HTTP
- recolectar_todo.sh: Ejecuta todos los monitores simultáneamente y detiene con Ctrl+C



Shell
Scripting
made Easy!

Complete guide for understanding basic
shell scripting syntaxes

```
#!/bin/bash

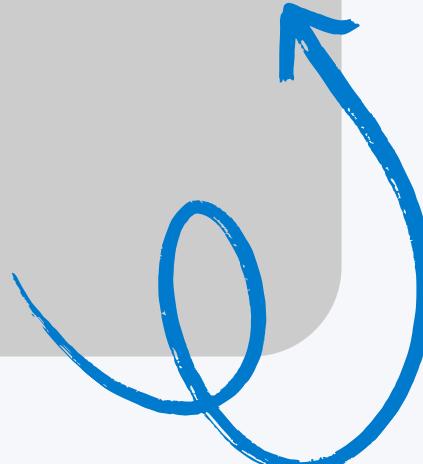
# declare a associative array
declare -A colours

# add key value pairs
colours['apple']=red
colours['banana']=yellow
colours['orange']=orange
colours['guava']=green
colours['cherry']=red

echo "Size of dict: ${#colours[@]}"
echo "Color of apple: ${colours['apple']}"
echo "All dict keys: ${!colours[@]}"
echo "All dict values: ${colours[@]}"

# Delete cherry key
unset colours['cherry']
echo "New dict: ${colours[@]}"

# iterate over keys
for key in ${!colours[@]}
do
    echo $key
done
```



Scripts de generación de carga

1.load_test.py:

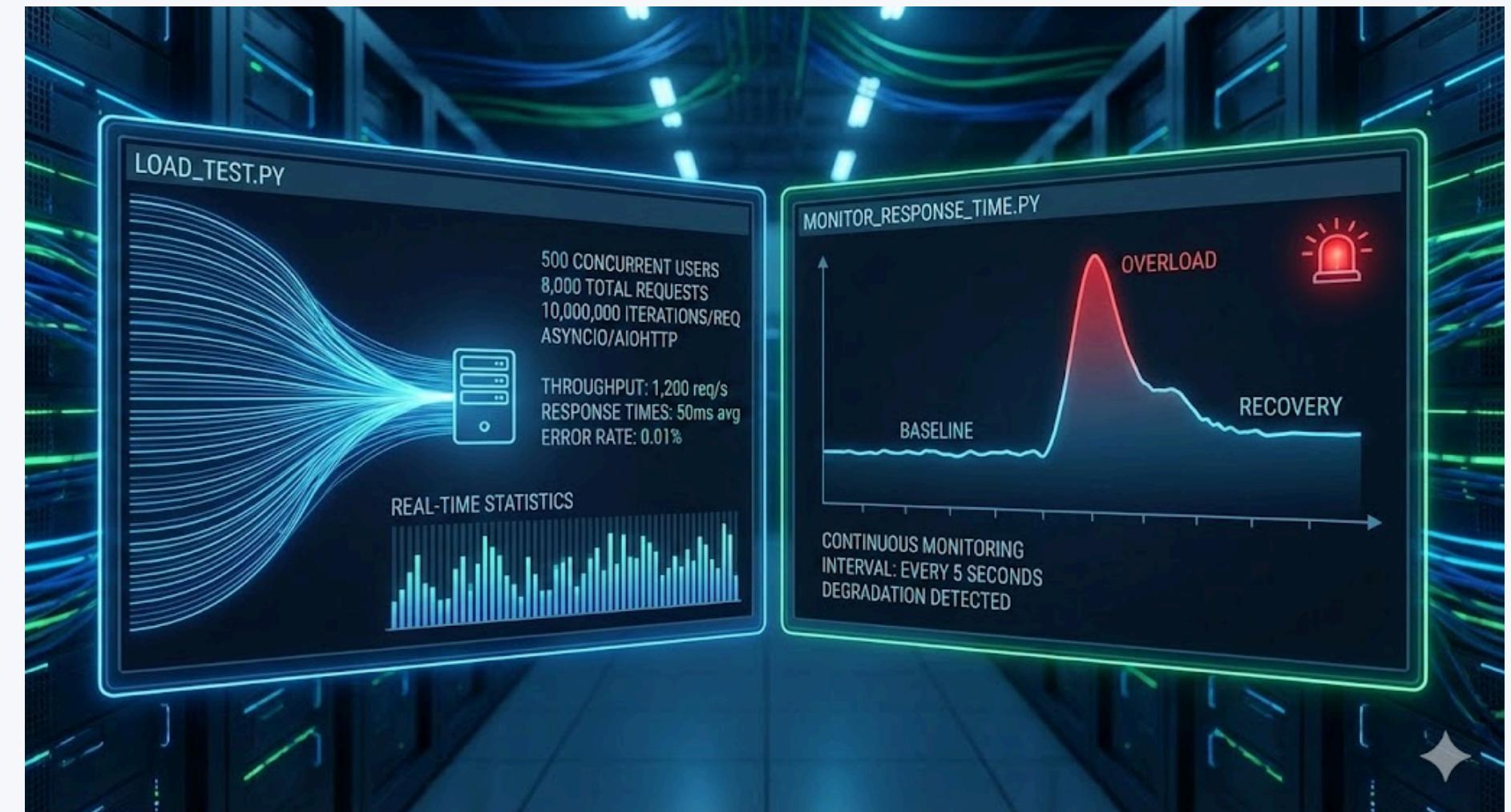
- Simula múltiples usuarios concurrentes.
- Utiliza asyncio y aiohttp para requests asíncronos.
- Configuración: 500 usuarios, 8.000 requests totales, 10.000.000 iteraciones por request.
- Genera estadísticas: throughput, tiempos de respuesta, tasa de error.

2.load_test_gradual.py:

- Simula múltiples usuarios concurrentes.
- Implementa un patrón de ramp-up que incrementa progresivamente la carga.
- Genera estadísticas: throughput, tiempos de respuesta, tasa de error.

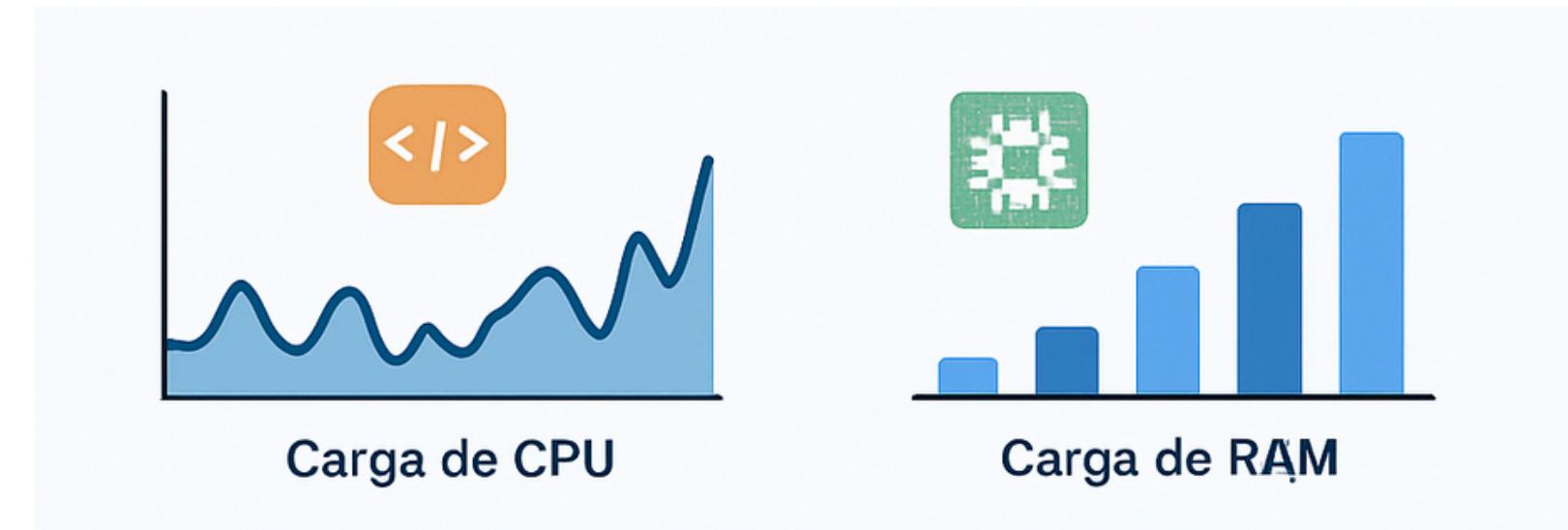
3.monitor_response_time.py:

- Monitoreo continuo de tiempos de respuesta.
- Permite capturar degradación (baseline → sobrecarga → recuperación).
- Intervalo: cada 5 segundos.



Implementacion de la aplicacion web

FastAPI actúa como generador controlado de carga de CPU y RAM para las pruebas. Expone endpoints deterministas que realizan operaciones matemáticas y de asignación de memoria con complejidad $O(N)$ para ajustar el esfuerzo de cómputo y consumo de memoria de manera lineal y proporcional.



Endpoints

- **GET /health:** Health check
- **GET /cpu:** Generador de carga CPU y RAM
- **GET /stress:** Endpoint principal para pruebas de carga que estresa CPU, memoria y red simultáneamente.

1. Carga de CPU:

- Operaciones de punto flotante (sqrt)
- Operaciones trigonométricas (sin, cos, tan)
- Complejidad temporal: $O(N)$ lineal con $N = \text{iteraciones}$

2. Carga de RAM:

- Almacenamiento de diccionarios con resultados intermedios
- Cada diccionario contiene 10 campos (valores calculados + duplicados)
- Complejidad espacial: $O(N)$ lineal con $N = \text{iteraciones}$
- Consumo estimado: ~400 bytes por elemento
- La memoria se libera automáticamente al finalizar el request

Metodología de prueba

- Prueba ramp-up en 5 fases (Baseline → Saturación).

Fases del Ramp-Up:

Fase	Usuarios	Duración	CPU	RAM	RED
Baseline	10	30s	100K	5MB	256KB
Moderada	50	30s	300K	10MB	512KB
Alta	100	30s	500K	15MB	768KB
Sobrecarga	200	60s	750K	20MB	1MB
Saturación	500	60s	1M	25MB	1MB

Metodología de prueba

- Monitores cada 5s: cpu, net, io, latency.

Scripts de Monitoreo

Script	Frecuencia	Métricas	Archivo de Salida
<code>monitor_cpu.sh</code>	5s	CPU (usr/sys), Load Avg, RAM	<code>cpu_metrics.csv</code>
<code>monitor_io.sh</code>	5s	IOPS lectura/escritura	<code>io_metrics.csv</code>
<code>monitor_net.sh</code>	5s	Bytes RX/TX	<code>net_metrics.csv</code>
<code>monitor_latency.sh</code>	5s	Tiempos HTTP	<code>latency_metrics.csv</code>

Metodología de prueba

- Resultados en CSV → graficado y análisis.

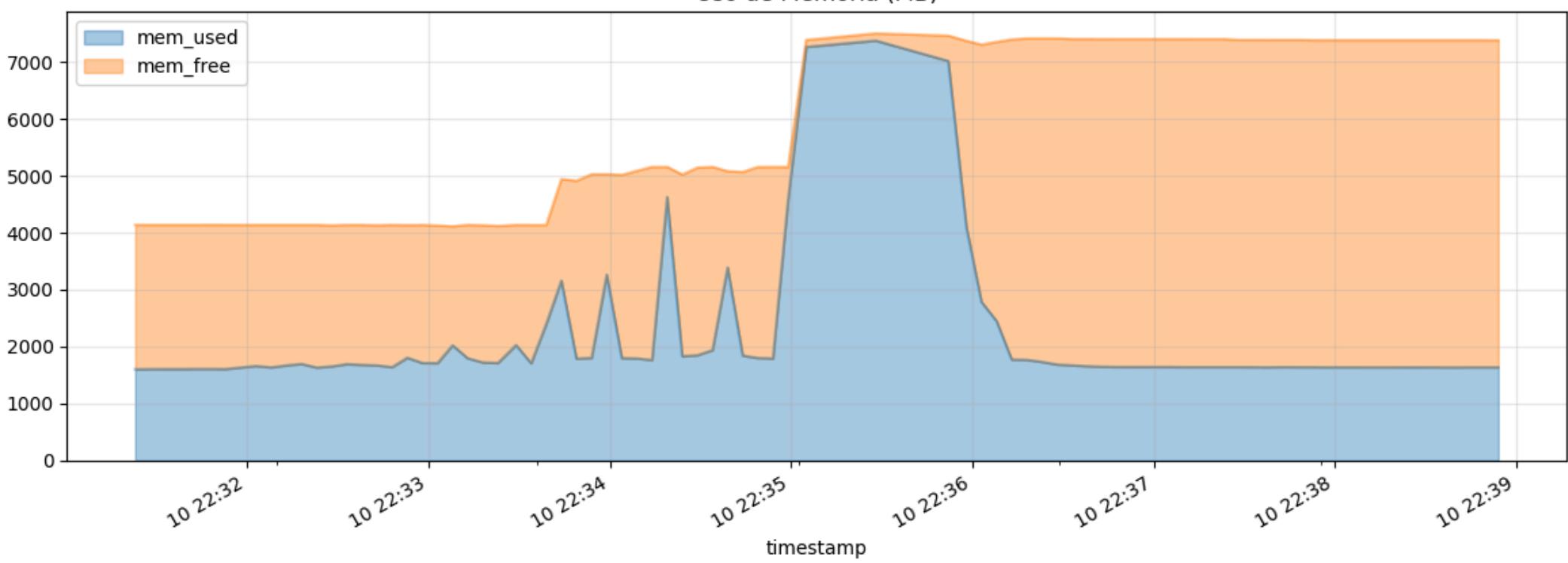
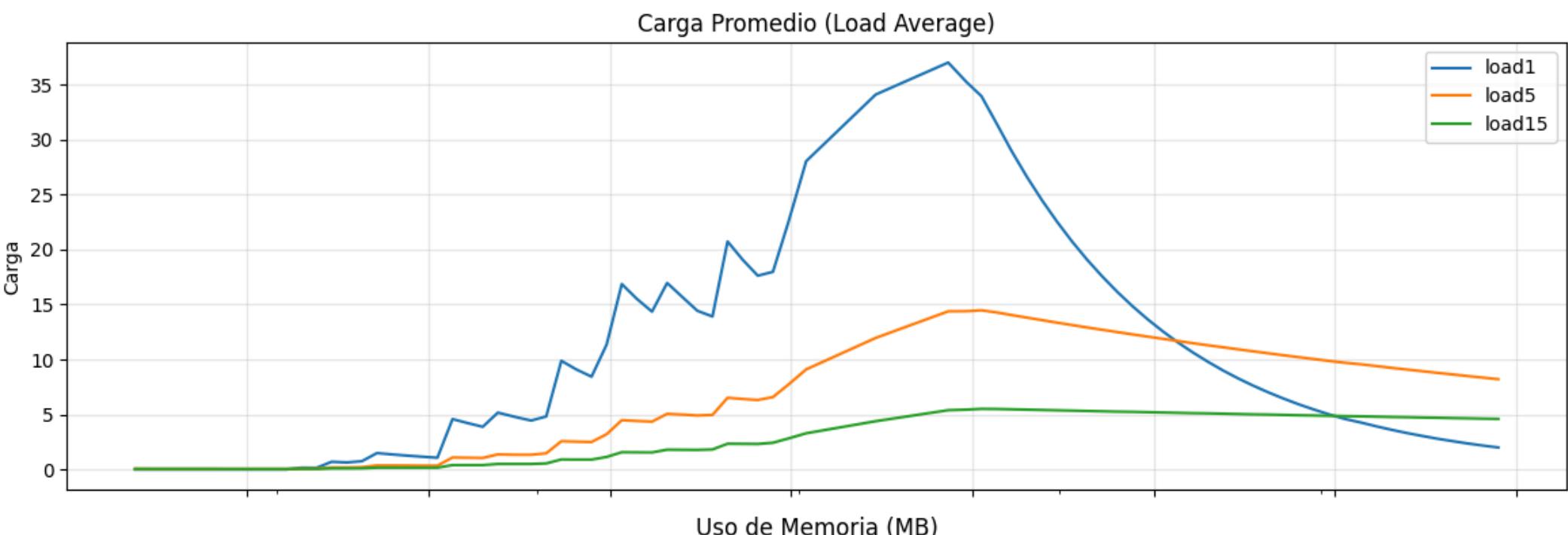
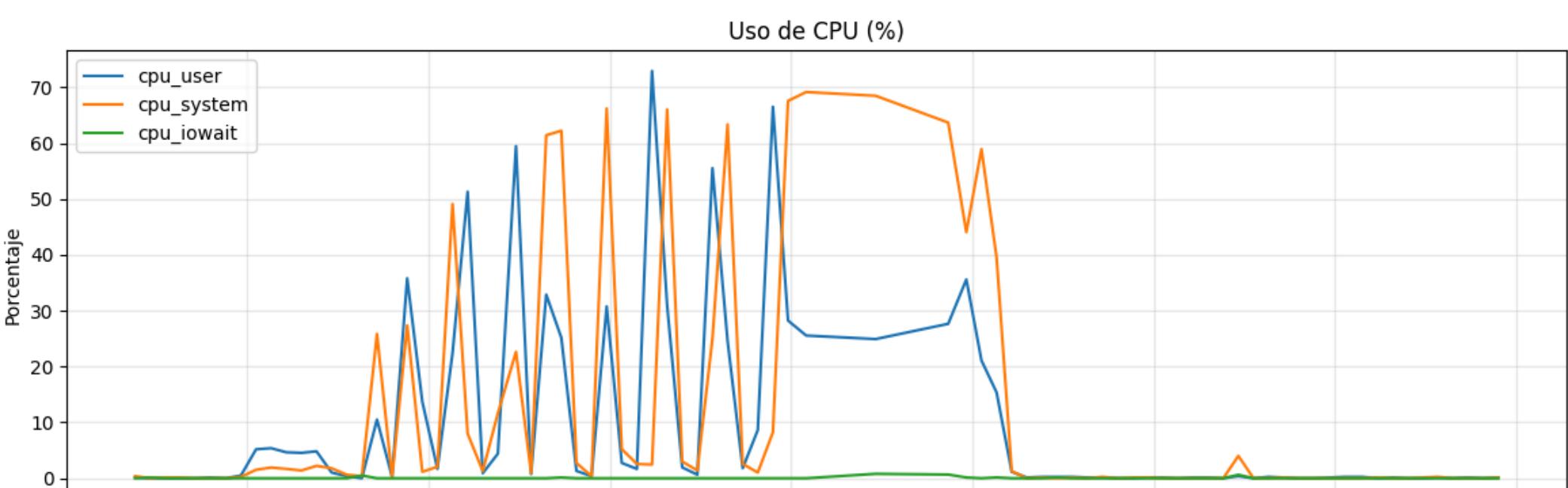
Gráficos Generados	
Archivo	Contenido
1_metrics_cpu_memoria.png	CPU, Load Average y Memoria
2_metrics_latencia.png	Latencia HTTP
3_metrics_red.png	Tráfico de Red (KB/s)
4_metrics_disco.png	Actividad de Disco
5_load_test_results.png	Resultados prueba de carga
6_response_time_metrics.png	Tiempos de respuesta
7_load_test_gradual.png	Resultados por fase (ramp-up)



Visualización de datos

Realizado con python y Matplotlib, permitía ver:

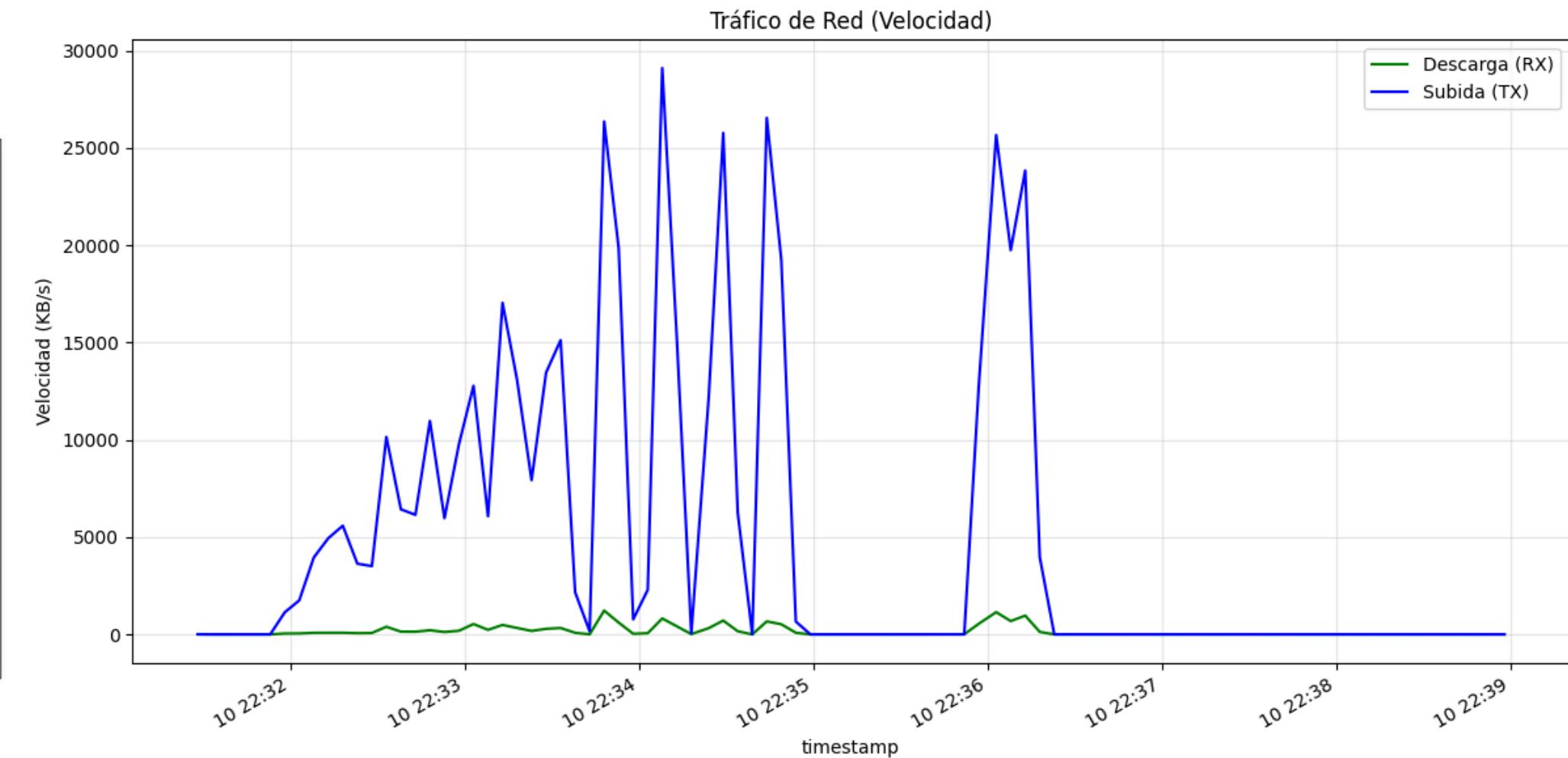
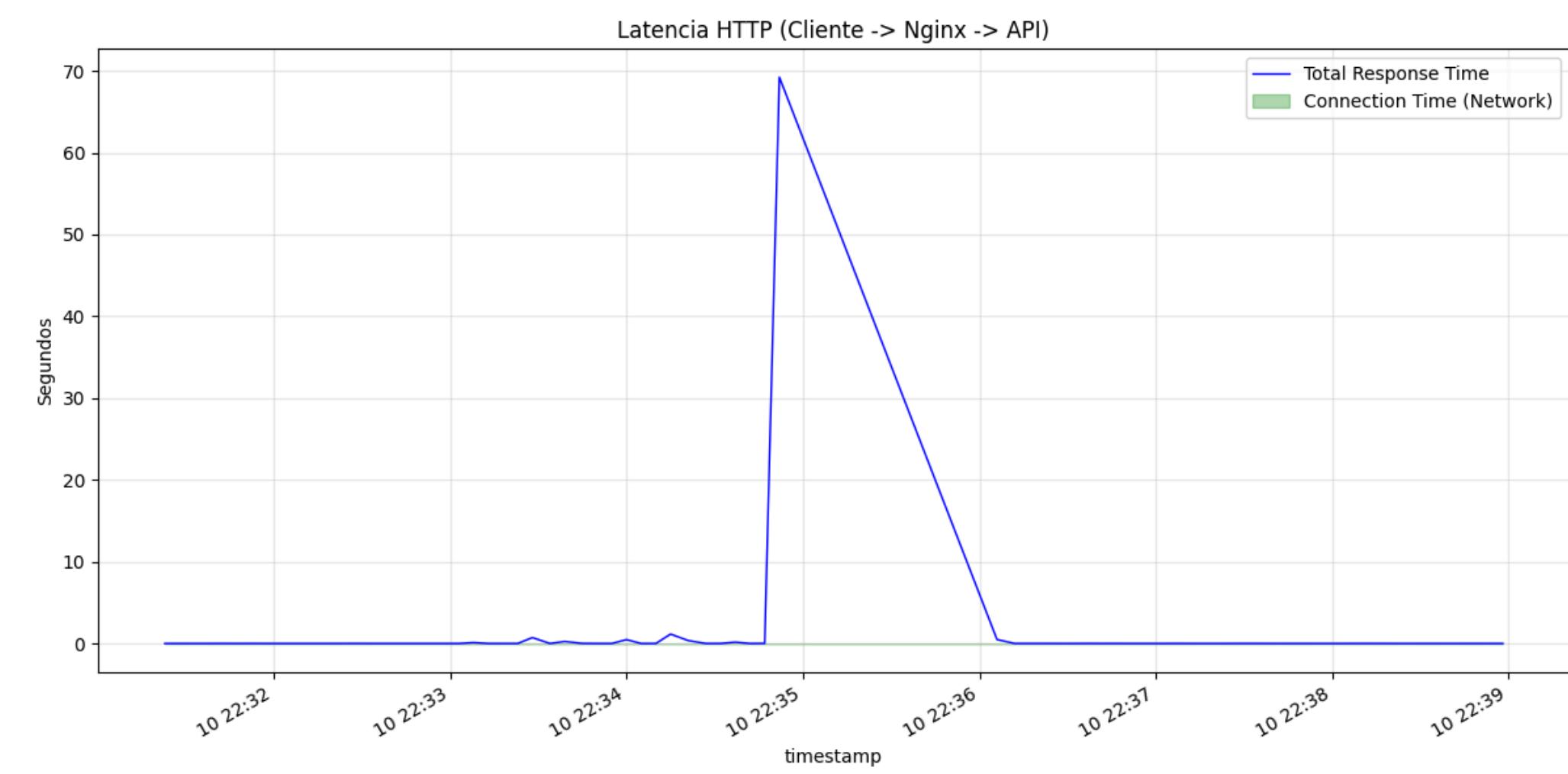
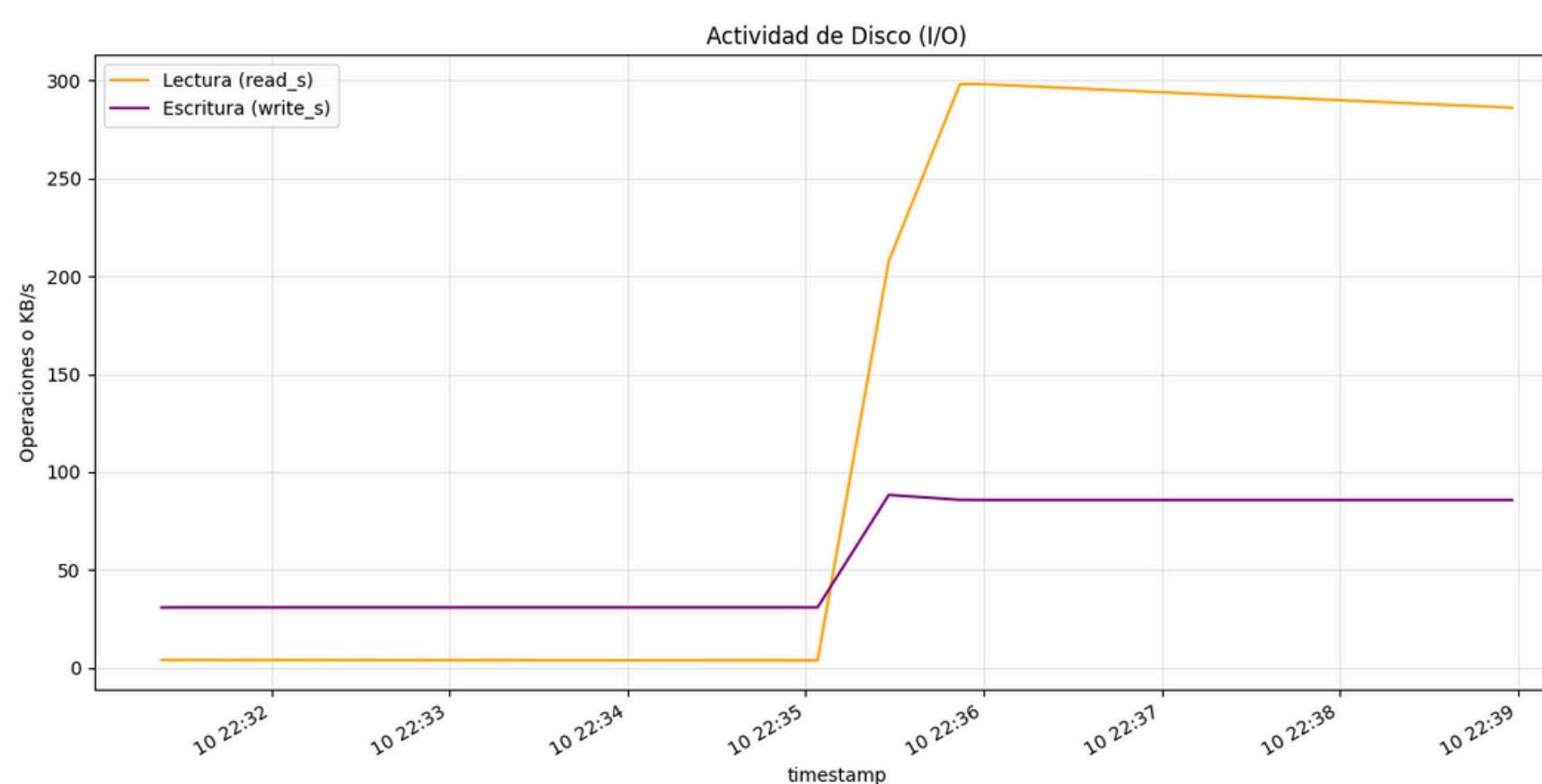
- Leer series temporales de archivos CSV.
- Generar gráficos de:
 - Carga del sistema (CPU, memoria, load average).



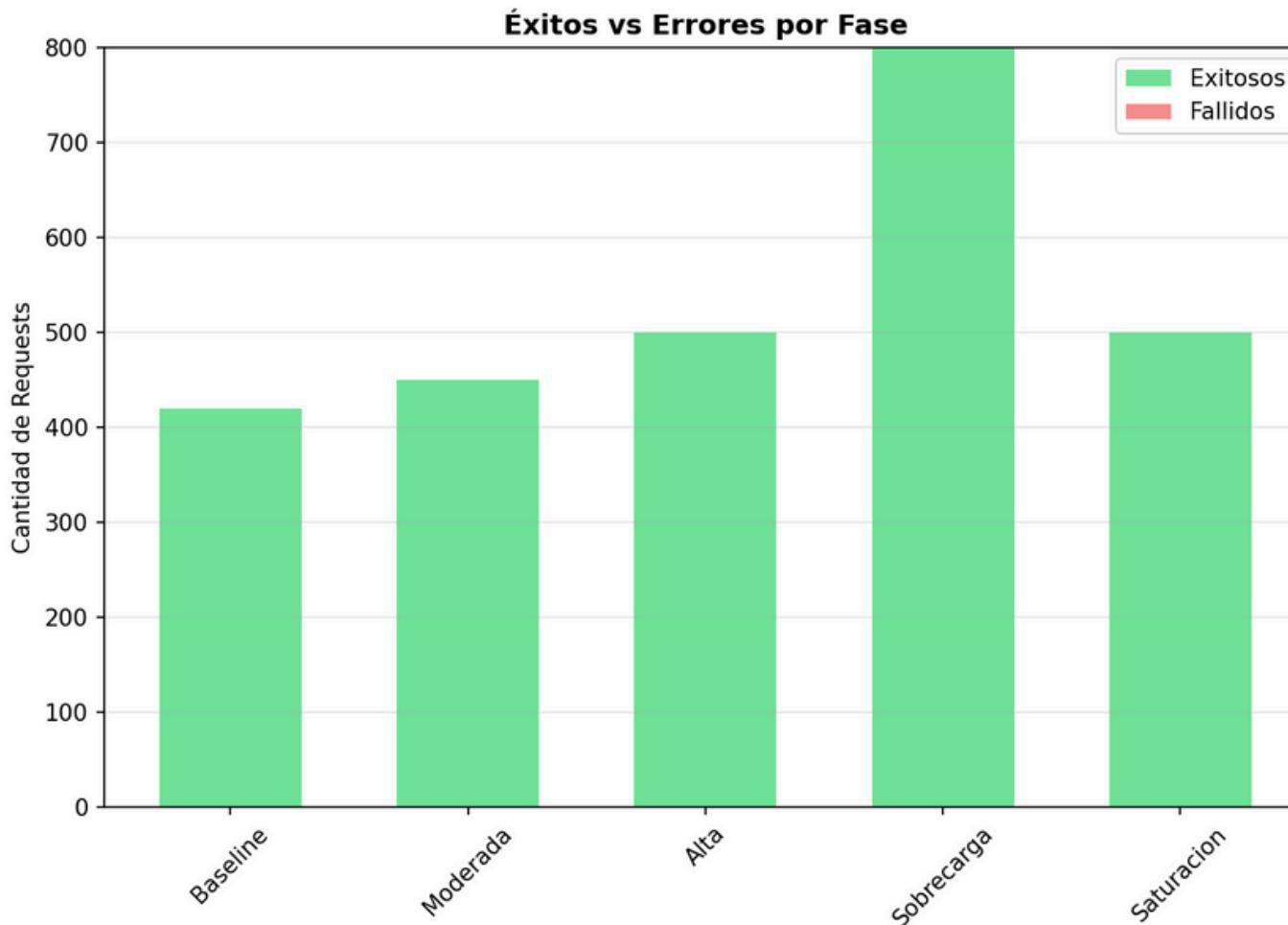
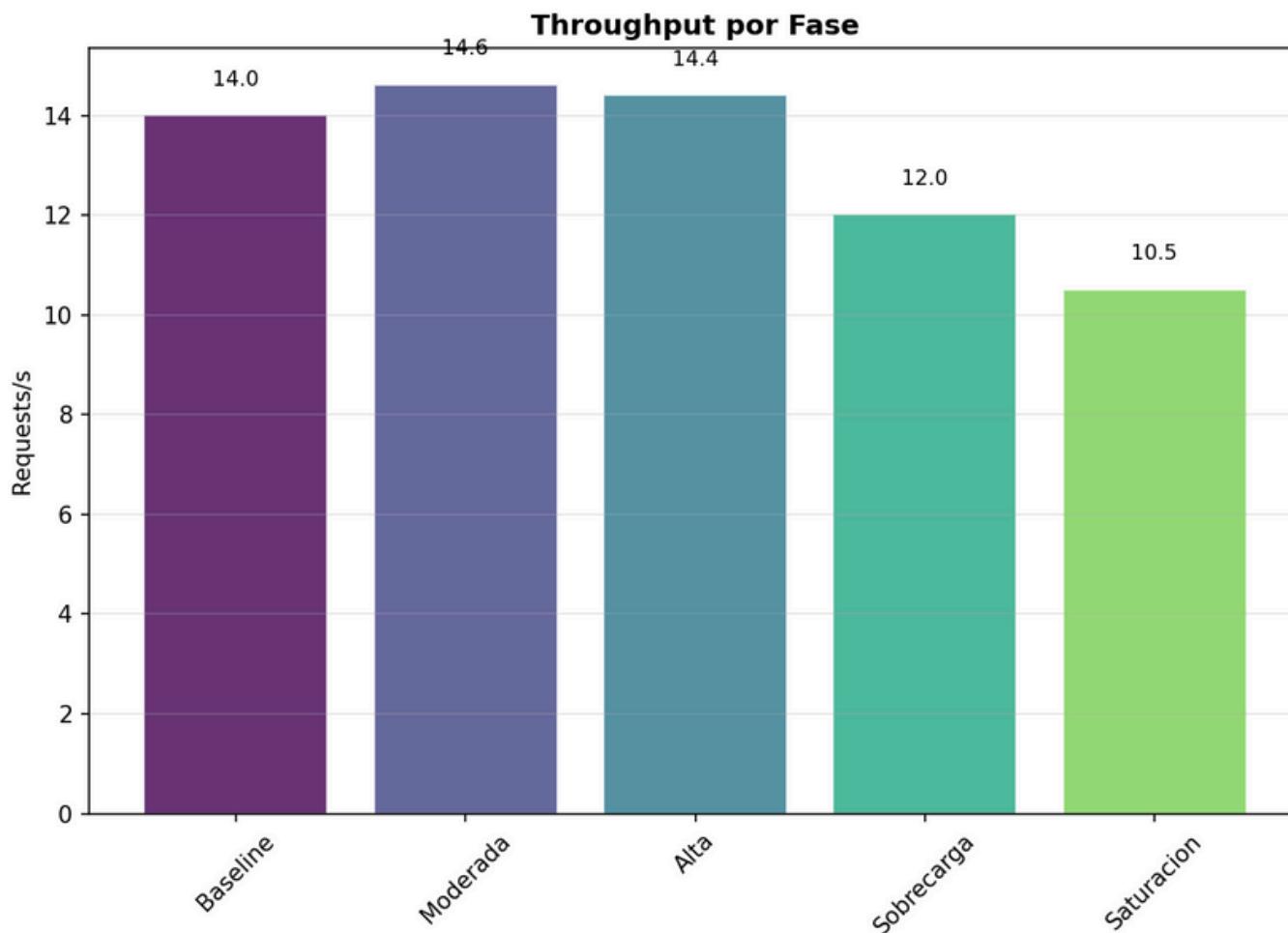
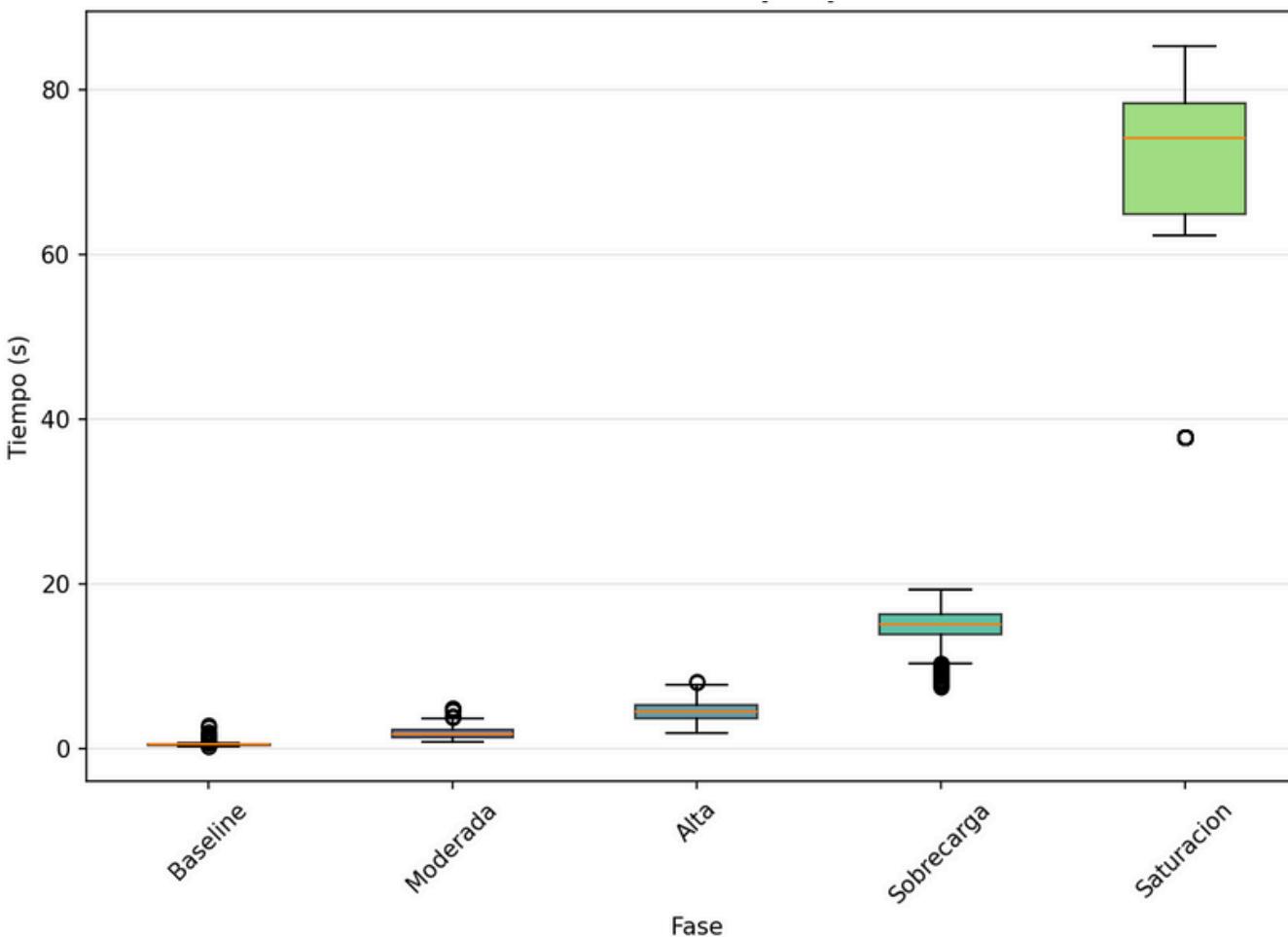
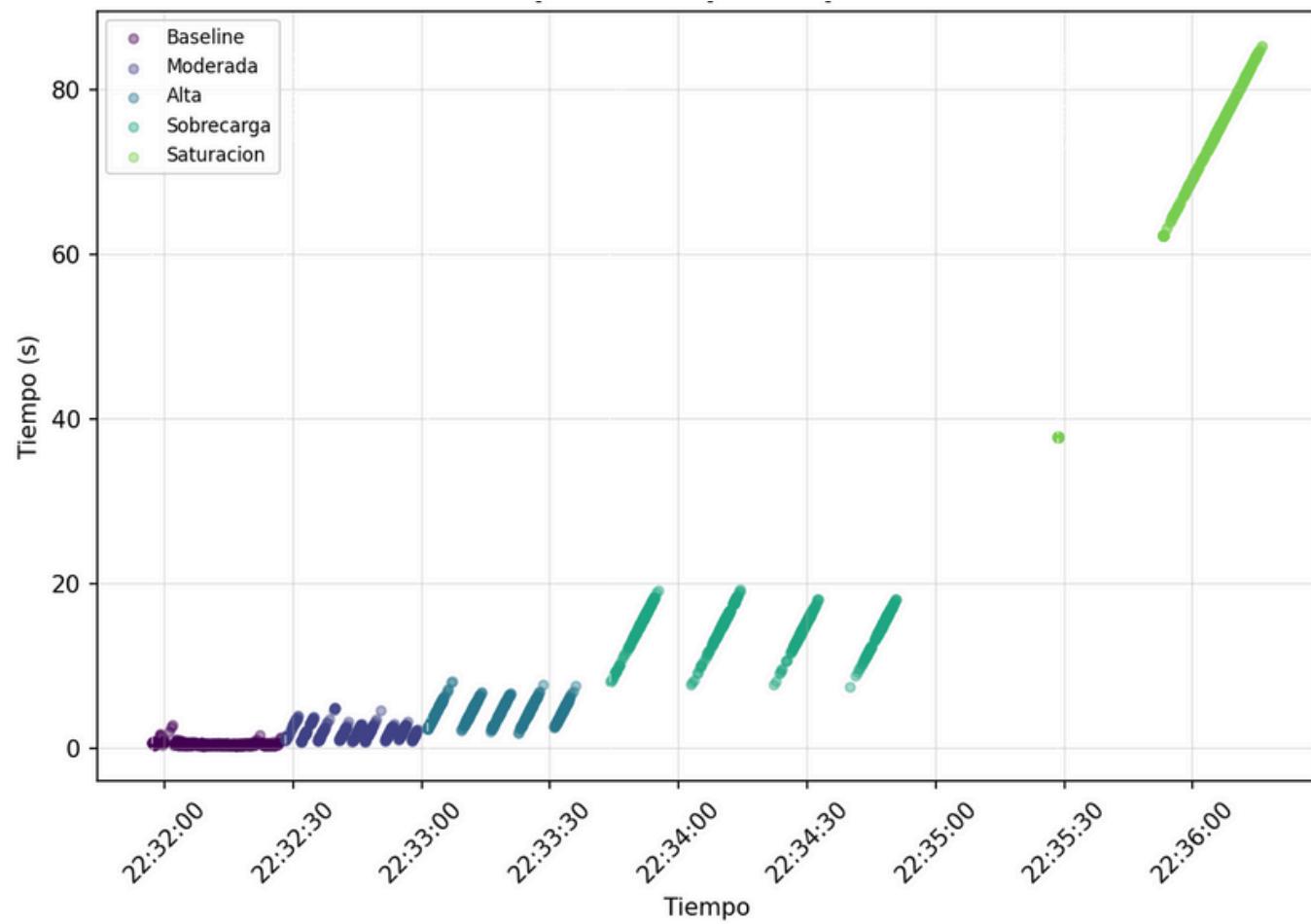
Visualización de datos

Realizado con python y Matplotlib, permitía ver:

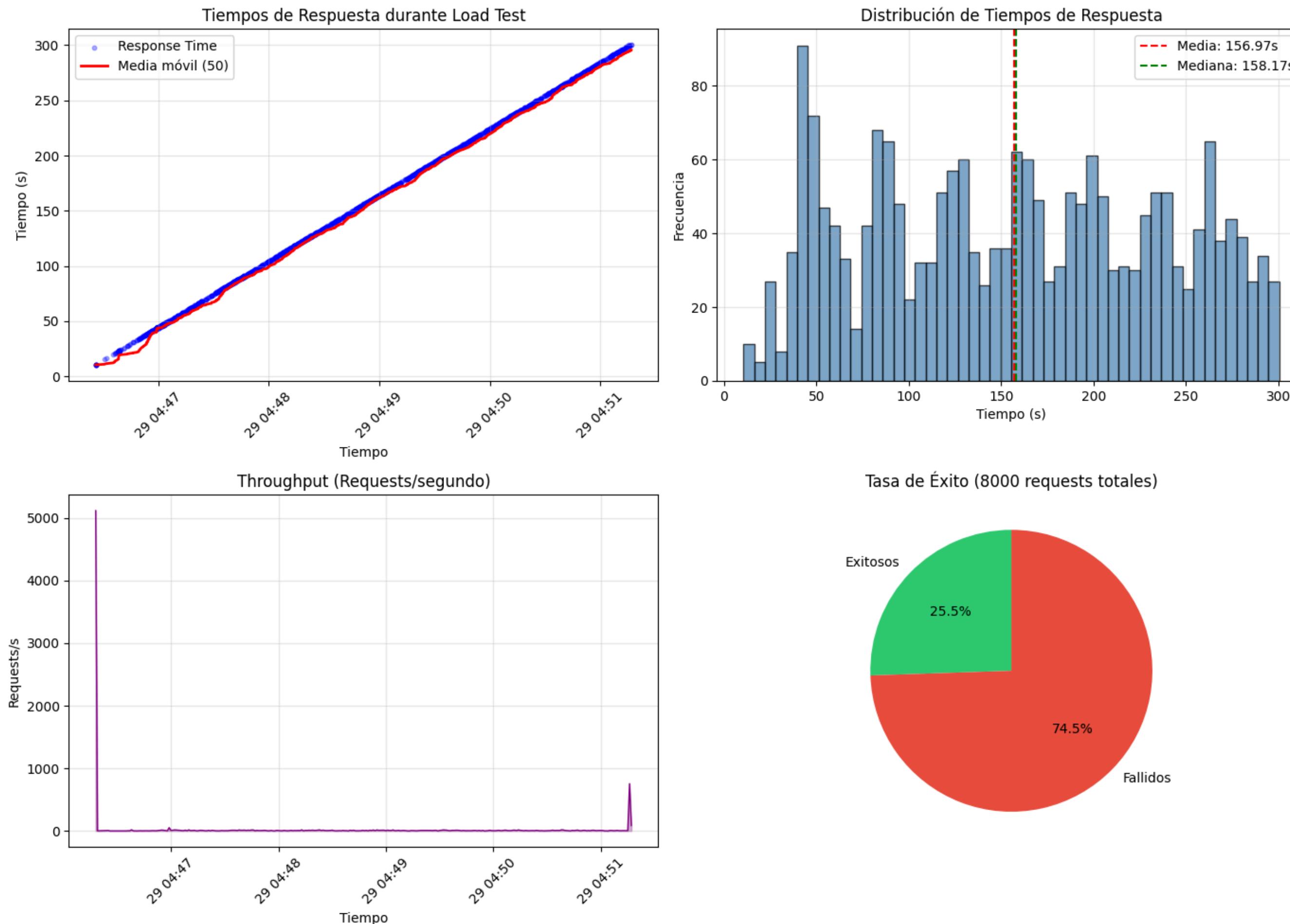
- Leer series temporales de archivos CSV.
- Generar gráficos de:
 - Latencia de respuesta.
 - Tráfico de red.
 - I/O de disco.
 - Throughput
 - Tiempo de Respuesta
 - Tasa de Éxito



Load Test Gradual



Load Test



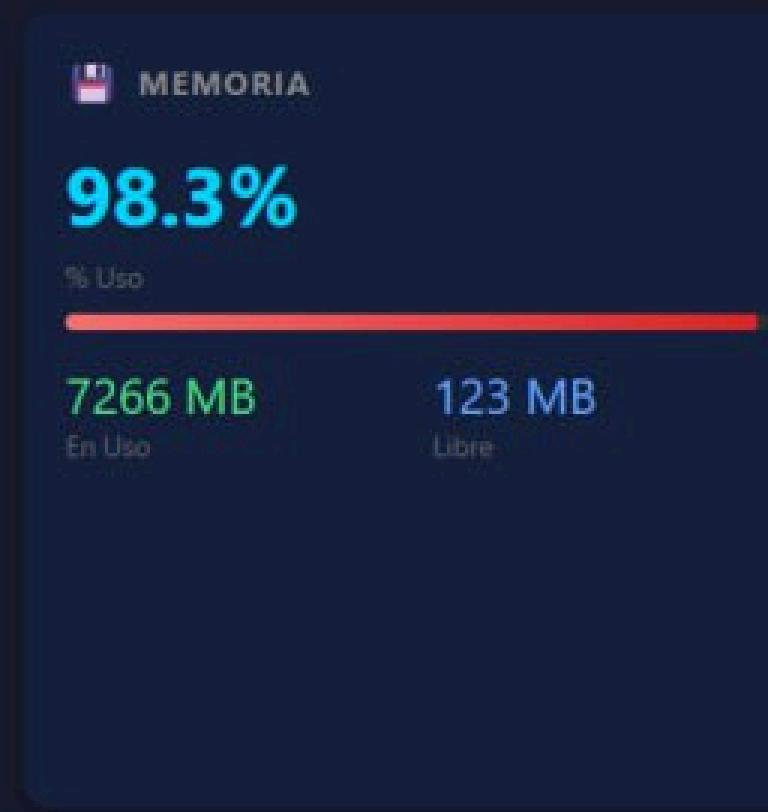
Visualización de datos



Visualización de datos

Dashboard de Monitoreo

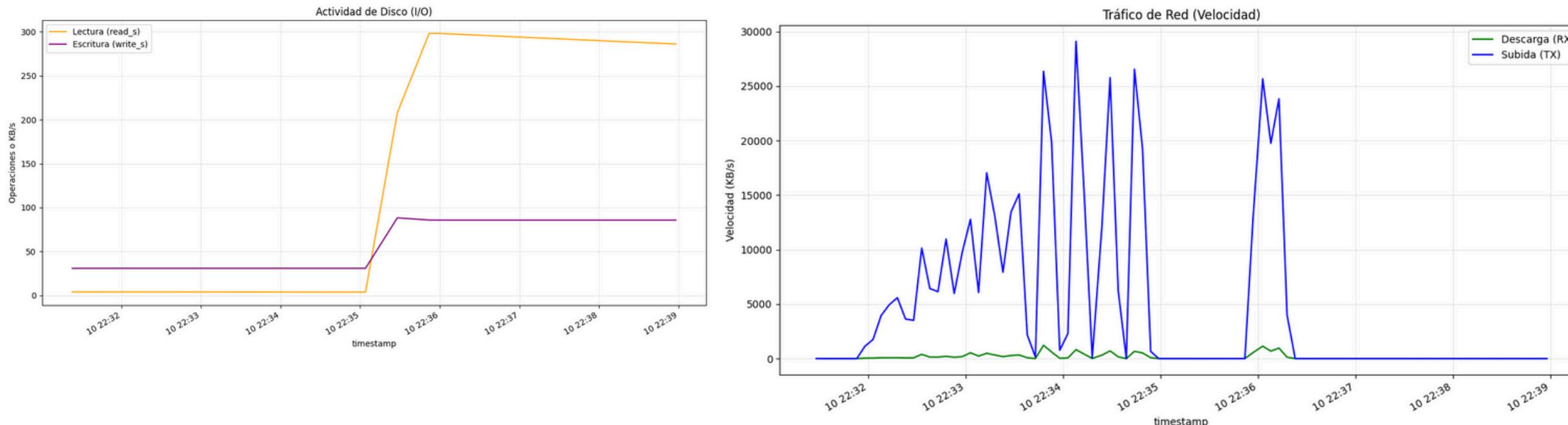
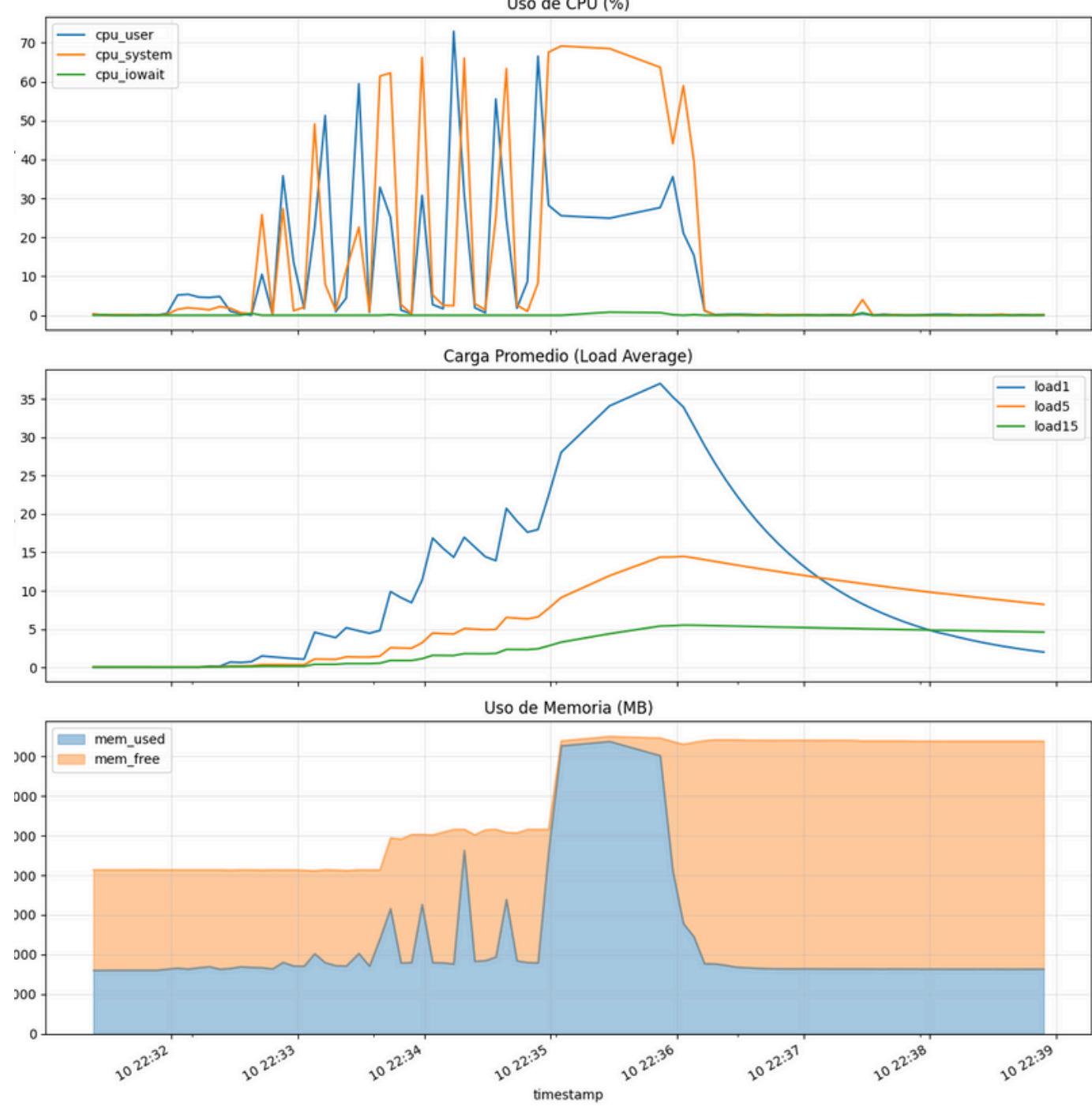
Conectado



Última actualización: 6:35:28 p. m.

Análisis

- Hasta la fase Alta, el servidor mantiene latencias bajas y estables.
- En Sobrecarga y Saturación, la p95/p99 de latencia crece exponencialmente.
- CPU alcanza 90–100% antes de que la latencia se dispare → primer cuello: CPU-bound.
- Luego aumentan IOPS e iowait, indicando un segundo cuello I/O-bound.
- En saturación, TX llega al límite y presenta caídas → cuello de red.
- Las métricas muestran correlaciones claras:
 - CPU ↔ Memoria ↔ Latencia
 - TX ↔ Latencia
 - IOPS ↔ Latencia
- El servidor entra en saturación multirecurso y ya no puede seguir aumentando el throughput, por lo que las solicitudes se encolan y la latencia se vuelve inaceptable.



Baseline y Carga Moderada

Puntos Clave:

- Baseline (1 usuario): Sistema en reposo, CPU menor al 5%, latencia de 60ms y sin overhead.
- Carga Moderada (10 usuarios): Escalamiento estable, CPU cerca del 50% y latencia de 300ms.
- Conclusión: El planificador reparte el trabajo sin fallas.



Escenario de Sobrecarga (Estrés)

Condiciones: 50 usuarios concurrentes.

- Impacto:
 - Saturación: CPU al 98% (Cuello de botella principal).
 - Colas: Load Average de 12.0 (vs 8 núcleos disponibles).
 - Degradación: Latencia explota a 2,500 ms (picos de 5s).
 - Fiabilidad: Tasa de éxito cae al 92% (Errores de conexión).



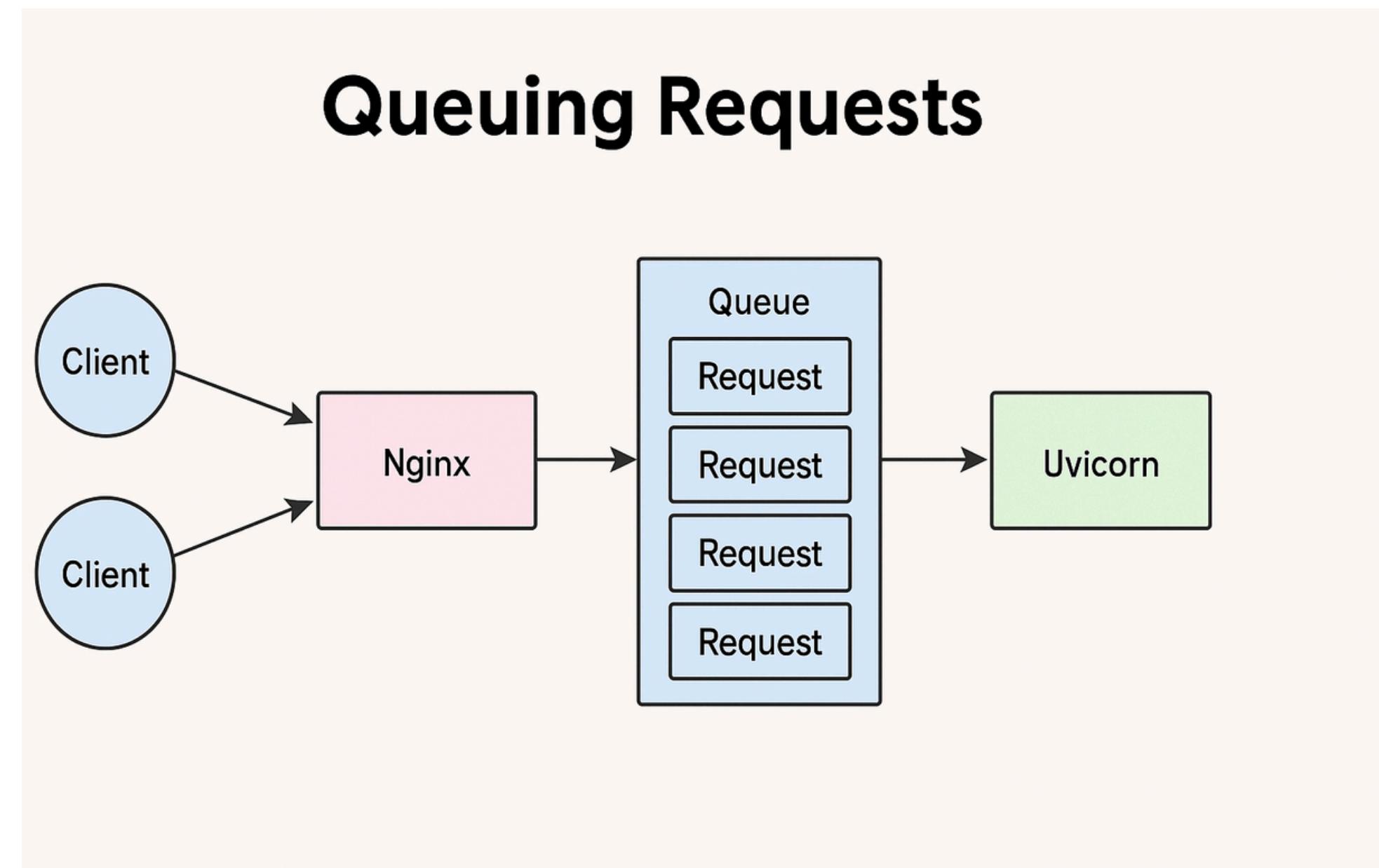
Throughput y Tasa de Éxito

- Métricas:
 - Capacidad máxima teórica: ~20 req/s.
 - Saturación efectiva al 87.5% de capacidad.
 - Errores (Sobrecarga):
 - 5% Fallos / 3% Timeouts.
- Causas: Nginx Gateway (502) y rechazo de conexiones.
- Lección: El throughput deja de escalar linealmente cuando Usuarios > Workers.



Plan de contingencia / medidas para evitar saturación

- 1.- Proceso de encolamiento de solicitudes
- 2.- Nginx con limit_req
- 3.- Respuestas degradadas (servir cache/estático)

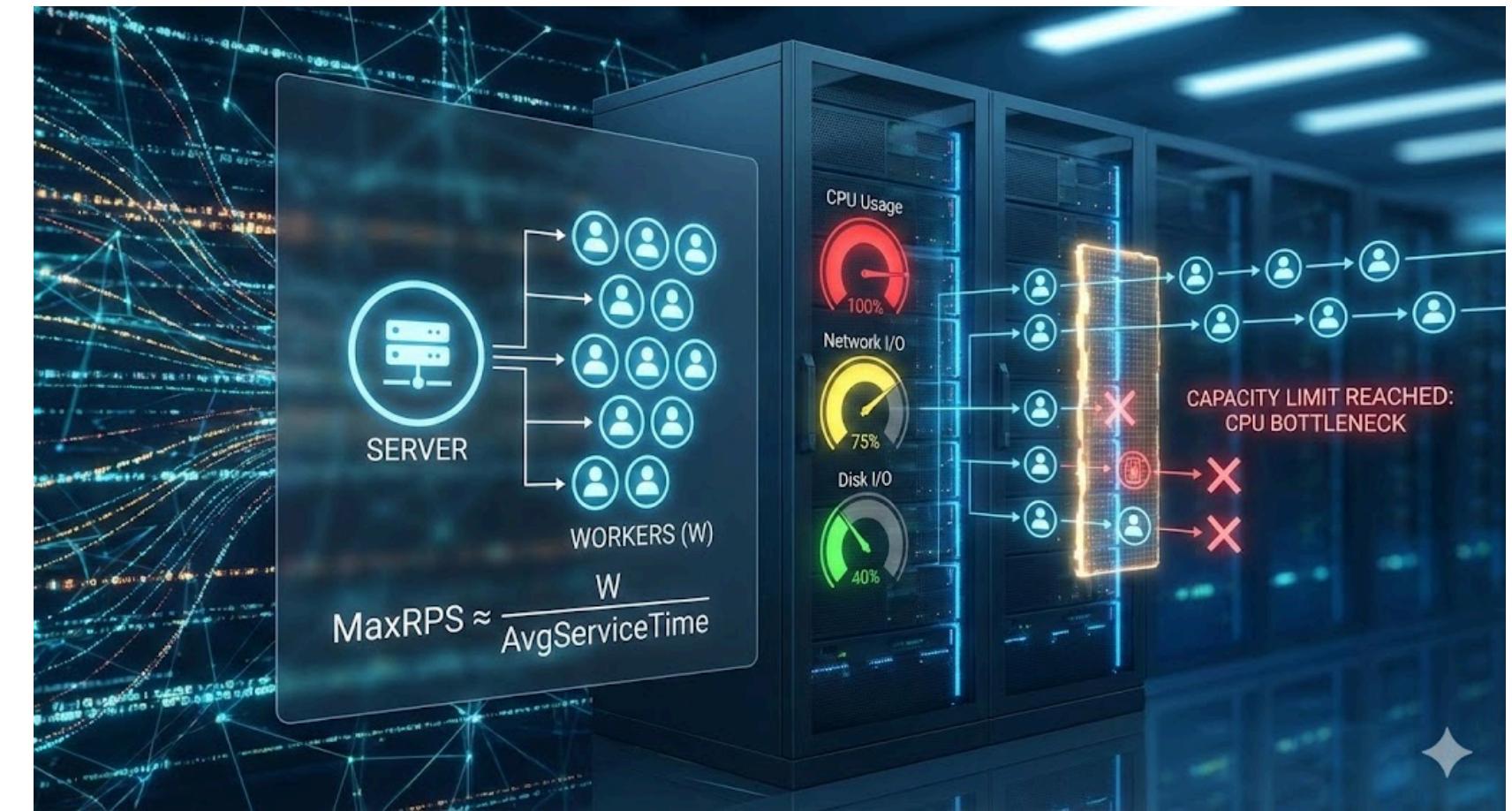


Monitoreo y Alerta

- Métricas de alerta:
 - **CPU > 75%**
 - **p95 latency > 500 ms**
 - **error rate > 1%**
 - **RX/TX > 80% NIC**
 - **iowait > 10%**
- Agregar alertas para crecimiento de memoria persistente.
- Centralizar logs y conservar muestras de tcpdump para incidentes.

¿Cómo gestiona el servidor la cantidad de usuarios?

El servidor gestiona usuarios mediante un conjunto de workers (Uvicorn/Gunicorn) que atienden solicitudes. Cada worker atiende requests concurrentemente (async o bloqueante según la carga). Nginx actúa de frontera que acepta conexiones TCP y las balancea a los workers. La capacidad efectiva resulta de la combinación: número de workers eficiencia por worker versus latencia del servicio.



¿Cuántas conexiones y solicitudes puede soportar?

No hay un número fijo universal: se determina experimentalmente.

Procedemos así:

- Medimos AvgServiceTime bajo carga controlada;
- Configuramos W workers;
- Estimamos MaxRPS(Requests Per Second) $\approx W / \text{AvgServiceTime}$;
- Validamos con pruebas ramp-up.

Además verificamos límites del sistema (ulimit, NIC, IOPS).

El número real estará limitado por el primer recurso que alcance 100% (CPU, red o I/O).

Conclusiones

Un sistema sólido y bien probado asegura disponibilidad, baja latencia y resiliencia ante incidentes.