**📌 INFORME DE RENDIMIENTO Y OPTIMIZACIÓN DE INFRAESTRUCTURA TI.**

**Cliente:**  
**Autor:** Oscar Macias.  
**Fecha:** 24 de marzo del 2025.  
**Versión:** 1.0

**ÍNDICE**

[**1️⃣ RESUMEN EJECUTIVO**💡 3](#_Toc193881614)

[**2️⃣ INTRODUCCION 🚀** 4](#_Toc193881631)

[**3️⃣ METODOLOGÍA 📌** 5](#_Toc193881632)

[**4️⃣ ANÁLISIS Y RESULTADOS 📊** 6](#_Toc193881637)

[**5️⃣ RECOMENDACIONES Y PLAN DE ACCIÓN🏆** 14](#_Toc193881642)

[**6️⃣ CONCLUSIÓN 🎯** 15](#_Toc193881643)

**7️⃣ REPOSITORIO DE INFRAESTRUCTURA**  16

**8️⃣ PRÓXIMOS PASOS**🔜 17

[**9️⃣ REFERENCIAS TÉCNICAS ✅** 18](#_Toc193881644)

**1️⃣ RESUMEN EJECUTIVO**💡

Se presenta el análisis, donde se identificaron problemas críticos que afectan la estabilidad y eficiencia del sistema, junto con recomendaciones para mejorar su desempeño y reducir costos operativos.

**📉 Principales Problemas Detectados**

🚨 **Fallos en la carga:** El 93% de los intentos de acceso fallan ❌

⏳**Lentitud extrema:** Tiempos de respuesta de hasta 13 segundos 🕒🐢

📍 **Falta de monitoreo:** No hay herramientas para detectar fallos en tiempo real⚠

💾 **Infraestructura limitada:** La base de datos y el caché presentan sobrecargas.

**💼 Impacto en el Negocio**

⚠️ **Pérdida de clientes:** La mala experiencia afecta la retención de usuarios.

⚠️ **Riesgo de interrupciones:** El sistema no soporta picos de tráfico 🚨

⚠️ **Costos elevados:** La infraestructura actual no es eficiente ni escalable

**🏆 Soluciones Propuestas**

✅ Optimizar la infraestructura actual para mejorar el rendimiento sin aumentar costos.

✅ Implementar monitoreo y alertas para detectar problemas antes de que impacten a los usuarios.

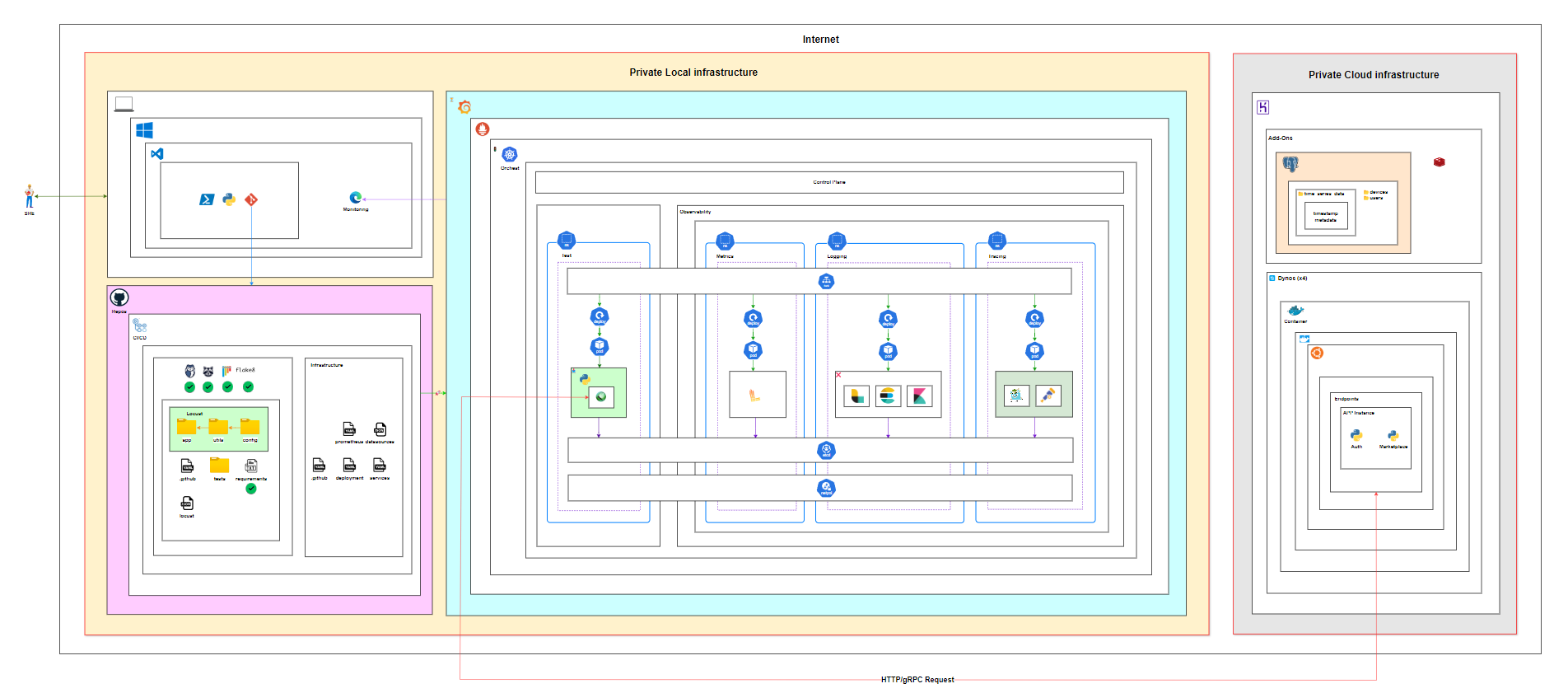
✅ Evaluar migración a **AWS** o **Render**, opciones más eficientes y rentables.

✅ Uso de CDN como **Cloudflare** o **AWS** **CloudFront** para mejorar tiempos de respuesta 📌

**2️⃣ INTRODUCCION 🚀**

Se evalúa la aplicación desplegada, incluyendo consumo de recursos, cuellos de botella y recomendaciones para escalabilidad, otras plataformas. Se comparan costos y beneficios, considerando recomendaciones concretas para su implementación en Datacenters de Europa.

**3️⃣ METODOLOGÍA 📌**

* **Estrés / Carga:** Desplegado en un Pod de Locust (Python) para Kubernetes
* **Metricas:** Grafana como visor del agente de Prometheus **📊**
* **Trazas:** Jaeger (Golang), utilizando los spas capturados por OpenTelemetry
* **Logs y monitoreo:** Pila ELK, donde Kibana visualiza la ingesta de Logstash, cargada y transformada por Elasticsearch. Loki como agente de Grafana⚠

👉 Se analizaron los tiempos de respuesta, tasas de fallos y la eficiencia del sistema actual.

**4️⃣ ANÁLISIS Y RESULTADOS 📊**

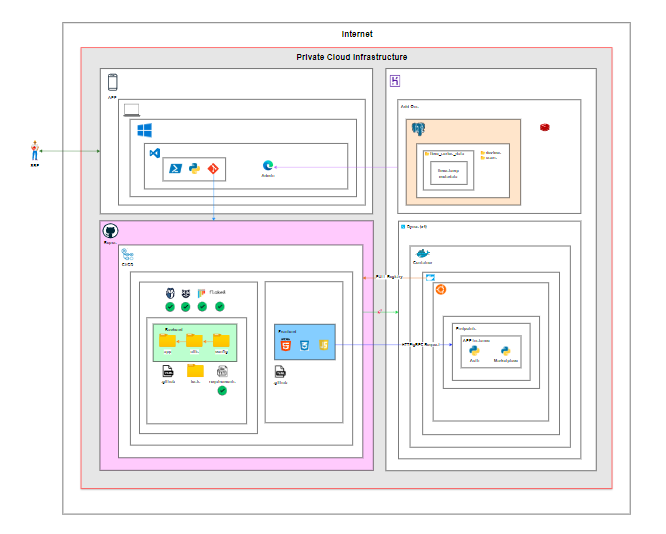
|  |  |
| --- | --- |
| **Métricas Relevantes** | **Resultado** |
| 📈 **Ratio de generación de usuarios (Segundos)** | **0.9** |
| ⏳**Duración de la prueba (Segundos)** | **2K** |
| ⚡**RPS alcanzado** | **93.2** |



📌 **Capacidad actual:** La carga no alcanzó los 1000 usuarios esperados, y el sistema no superó el caching. Se requiere optimización.

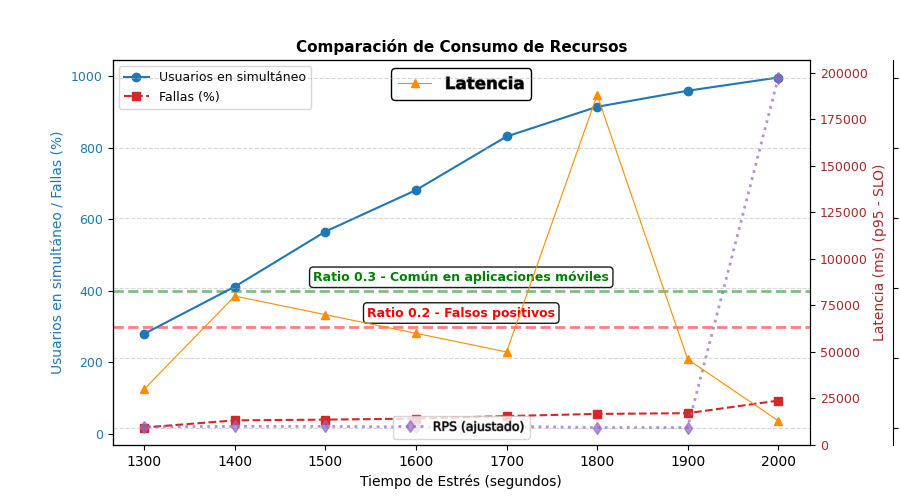
**Infraestructura Actual en Heroku 🏗**

* **Backend:** Desplegado en un contenedor Docker 🐳
* **Base de datos:** PostgreSQL, Bloqueos en queries concurrentes 🛢
* **Caching:** Redis, Expulsión de datos por falta de memoria 🔥
* **Proxy/API Gateway:** Sin control de tráfico, sobrecarga directa al backend🚫



**Análisis de Recursos y Costos 💰**

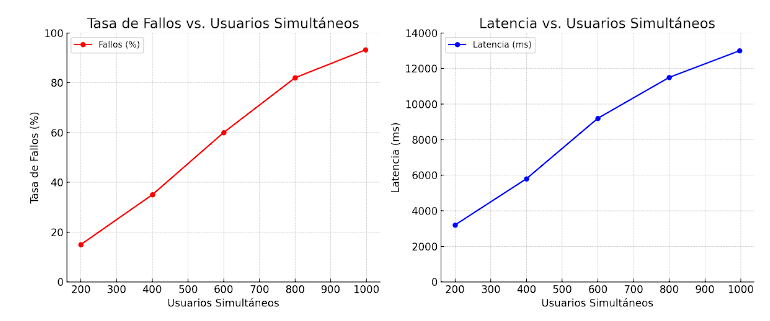
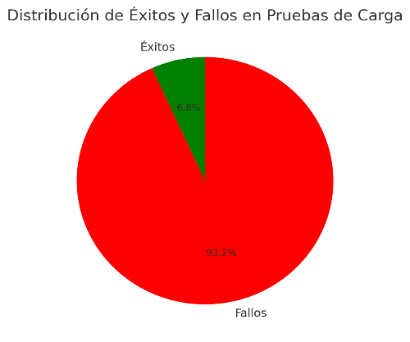
|  |  |  |
| --- | --- | --- |
| **Recurso** | **Estado Actual en Heroku** | **Requerimientos >= 1000 usuarios constantes** |
| 💾 **Dynos** | **x4** 🔄 | **Optimizar Gunicorn y RAM** 🏗 |
| ⚡ **Redis** | **Add-on limitado** 🛑 | **Escalar a mayor capacidad** 📈 |
| 🛢 **PostgreSQL** | **Sin clustering** ❌ | **Evaluar particionado** 🔄 |



📉 **Análisis:** La tasa de fallos aumenta drásticamente a medida que crece la cantidad de usuarios, indicando que la infraestructura actual no escala adecuadamente.

**📊 Visualización del problema:**

|  |  |  |
| --- | --- | --- |
| **Usuarios Simultáneos** 👥 | 🟠 **Latencia en percentil 95 (K / ms)** 📈 | 🔹 **Tasa de fallos (%)** ❌ |
| **200** | **3.2** | **15** |
| **400** | **5.8** | **35** |
| **600** | **9.2** | **60** |
| **800** | **11.5** | **82** |
| **997** | **13** | **93.2** |



**📌 Sugerencia de Valores Base para Comparación**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Métrica** | **Usuarios** | | | | |
| **100** | **250** | **500** | **1K** |
| **Fallas Aceptables (%)** | **2** | **3** | **5** | **7** |
| **RPS Esperado** | **100** | **250** | **500** | **1K** |
| **Tiempo de Estrés (minutos)** | **10** | **15** | **20** | **30** |
| **Latencia P95 (ms)** | **< 200** | **< 300** | **< 500** | **< 700** |

**🟠 Dimensionamiento de PostgreSQL (Heroku Add-on, sin clúster)**

**🔢 *Fórmula para las conexiones necesarias***

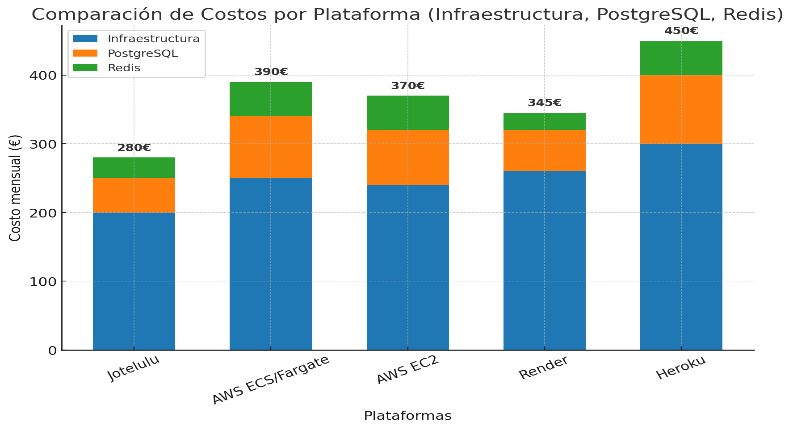
|  |  |
| --- | --- |
| **Conexiones necesarias=** | **(Usuarios simultáneos​)** |
| **(Factor de conexion por usuario​)** |

* + **Factor de conexión por usuario:** 5 si hay buen caching, 1 si cada usuario genera una consulta nueva 🔹

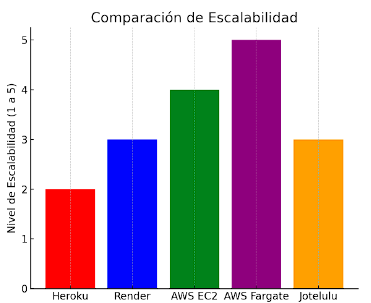
|  |  |
| --- | --- |
| **200 conexiones=** | **1000** |
| **5** |

**Comparación de Plataformas 📌**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Plataforma** | **Latencia 🚀** | **Escalabilidad** 🔄 | **Estimado (€ / mes) 💰** | **Uso 🔧** |
| **Jotelulu** | **Baja** ⚡ | Media ❌ | 280 | Fácil ✅ |
| **AWS ECS/Fargate** | **Muy Alta** 📈 | **390** | **Requiere configuración** 🛠 |
| **AWS EC2** 🔄 | **Alta** 📈 | **370** |
| **Render**  🔄 | Media ⚡ | **320** | **Fácil** ✅ |
| **Heroku** 🏗 | **450** |



***Nota: Estos valores deben validarse con cada proveedor según región.***

******

**Evaluación de Redis y PostgreSQL en Cada Escenario 🔍**

|  |  |  |
| --- | --- | --- |
| **Actual** | **Redis** | **PostgreSQL** |
| **Heroku** | **Add-on limitado** | **Add-on sin clustering** |
| **AWS EC2** | **Elasticache** | **RDS** 🔄 |
| **AWS ECS** |
| **Jotelulu** | VM | |
| **Render** 🆓 | **25 conexiones.** | **100 MB.** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Prospecto** | **Infraestructura Principal** | **Redis** | **PostgreSQL** |
| **Heroku** | **4 Dynos Standard 2X** 🔄 | **Premium 0** | **Standard 0** |
| **AWS EC2 (Memory-bound)** | **1x EC2 t3.medium (2vCPU, 4GB RAM)** 🔄 | **Amazon ElastiCache (cache.t4g.small)** | **Amazon RDS (db.t3.small, 100GB SSD)** |
| **AWS ECS** | **2x Fargate (CPU 0.5vCPU, 1GB RAM)** | **Amazon ElastiCache (cache.t4g.micro)** | **Amazon RDS (db.t3.micro)** |
| **Jotelulu (VMs Dedicadas)** | **1x VM (2 vCPU, 4GB RAM)** | **VM (1GB RAM)** | **VM (100GB SSD)** |
| **Render** | **2x servicios Standard (2GB RAM)** | **Standard (25MB)** | **Starter (50GB)** |

**5️⃣ RECOMENDACIONES Y PLAN DE ACCIÓN🏆**

✅ **Pruebas de carga post-optimización:** Validar mejoras.

📌 **Logging centralizado:** Para análisis post-mortem de fallas.

📌 **Alertas proactivas:** Ante problemas de rendimiento.

📌 **Tolerancia a Fallos y Recuperación (HA).**

**✅ Considerar probar add-ons útiles:** BlazeMeter (para pruebas de carga y tráfico) o Runscope (para monitoreo de APIs).

**6️⃣ CONCLUSIÓN 🎯**

📢 El presente es un punto de partida para la toma de decisiones estratégicas. Se recomienda priorizar la optimización para garantizar una plataforma estable, rápida y escalable para los usuarios 🚀

**7️⃣** **REPOSITORIO DE INFRAESTRUCTURA**

Para replicar la prueba, el código fuente y la configuración de la infraestructura utilizada están disponibles en **GitHub**:

**🔗** [**OMaciasd/load-testing-locust**](https://github.com/OMaciasd/load-testing-locust)

**Contenido del Repositorio:**

* **docker-compose.yml:** Definición de los servicios (Locust, InfluxDB, Grafana).
* **locustfile.py:** Escenarios de carga utilizados.
* **grafana-dashboard.json:** Configuración de paneles en Grafana.
* **README.md:** Instrucciones para levantar el entorno y ejecutar pruebas.

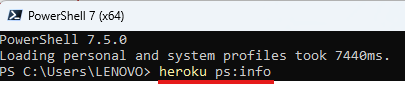
**8️⃣ PRÓXIMOS PASOS** 🔜

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Recurso** | **Pruebas pendientes en Heroku** | **Aplicado** | **Prueba de carga y estrés** | **Superado** |
| ⚡ **Redis** | 📌 **Configurar Premium 0.** |  |  |  |
| **Monitoreo y Alertas** | 📌 **Implementar herramientas.** |  |  |  |
| 🛢 **PostgreSQL** | 📌 **Configurar Standard 0.** |  |  |  |
| 💾 **Dynos** | 📌 **Medir impacto en latencia local dentro del Docker con herramientas de monitoreo.** |  |  |  |

**📌 Con estas acciones, evaluaremos una segunda opinión de un servicio más eficiente, confiable y rentable.**

**9️⃣ REFERENCIAS TÉCNICAS ✅**

**🔧 Optimizar Dynos con Gunicorn**

* Ajustar el número de **workers**, según la carga.
* ****Reducir RAM y mejorar eficiencia, ajustando el número de **threads**.
* Evitar **Memory Quota Exceeded (R14)**.
  + Usa el dashboard de Heroku o Heroku CLI para ver el consumo 📊

- Si ves **R14 (Memory Quota Exceeded)** o **H12 (Request timeout)**, es momento de ajustar workers o escalar dynos.

* + **Heroku mata dynos** si superan la RAM permitida 🎯

*# gunicorn.conf.py*

max\_requests = 1000

max\_requests\_jitter = 100

* Habilitar **Keep-Alive** para reducir conexiones innecesarias 🛠

*#gunicorn.conf.py*

keepalive = 5

timeout = 30

* + **1 dyno bien configurado** puede manejar **400 usuarios**.
  + Para **1000 usuarios**, podrías necesitar **4 dynos**, reduciendo el costo en un **60%.**

**Habilitar logs y monitoreo**

* Utilizar herramientas como **Papertrail** o **LogDNA** para errores en tiempo real.
* Implementar métricas en la aplicación (**Heroku Metrics**).
* Configurar un sistema de alertas para prevenir fallos críticos🔹

**🔥 Optimizar Redis y PostgreSQL**

* 🧩 Usar **menos claves y valores más compactos** **🚀**



* + Identificar consultas lentas con **EXPLAIN ANALYZE** 🔹
* Monitorear, e Indexar queries más utilizadas para evitar cuellos de botella.



* + **HSET** crea un hash en Redis con la clave **$user:$VALUE\_USER**.
  + Almacena los campos **$name** con sus respectivos valores.
* Reducir el **TTL** para cacheo ✅



* + **SET** almacena el valor en la clave **$home**.
  + **"<html>...</html>"** representa el contenido **HTML** que se guarda.
  + **EX 600** establece una expiración de **600 segundos (10 minutos)**.
  + Uso de **Read Replicas**.
  + Esto crea una réplica de solo lectura de la base de datos principal🔹
  + La aplicación puede distribuir lecturas entre réplicas para mejorar rendimiento🔹
* Usar pgbouncer para manejar conexiones y evitar que muchas consultas consuman recursos ✅
  + Si ves algo como **PgBouncer** en la respuesta, significa que la conexión está pasando a través de él.
  + Si la URL comienza con **postgres://** en lugar de **rediss://**, significa que **PgBouncer** está activo.

**💲 Alternativas de Ahorro de Costos**

* Implementar API Gateway (**Kong** / **Envoy (Kubernetes)**) para reducir carga.
* Habilitar **Preboot** para evitar reinicios costosos.



📌 **Usar HST en esta fase (Pruebas de Carga en Staging).**

* 📢 Para garantizar un lanzamiento exitoso, se recomienda integrar un sistema de detección de anomalías que identifique problemas de rendimiento en los tiempos de respuesta, uso de recursos y comportamiento de endpoints durante las pruebas de carga con Locust. Esto permitirá identificar cuellos de botella y problemas de escalabilidad antes del despliegue en producción.