**20-4-2025**

Oscar Andres Macias Narvaez

**Informe de Rendimiento y Optimización de Infraestructura TI**

**EVALUACIÓN DE ESCALABILIDAD, EFICIENCIA Y RECOMENDACIONES ESTRATÉGICAS.**

### CONTROL DE CAMBIOS⚖️

|  |  |  |  |
| --- | --- | --- | --- |
| 📄**Versión** | 🗓️**Fecha** | ✏️**Autor** | 🛠️**Cambios Relevantes** |
| **1.0** | **24-mar-25** | **Oscar Macias** | **Análisis inicial de infraestructura Heroku** |
| **2.0** | **20-abr-25** | **Actualización de stack** |

### CAMBIOS ESPECÍFICOS EN ESTA VERSIÓN📃

* Ajuste del stack tecnológico reportado: Ruby on Rails con Puma.
* Inclusión de RabbitMQ y Sidekiq.

**ÍNDICE**

[**1️⃣ RESUMEN EJECUTIVO**💡 4](#_Toc193881614)

[**2️⃣ INTRODUCCION 🚀** 5](#_Toc193881631)

[**3️⃣ METODOLOGÍA 📌** 6](#_Toc193881632)

[**4️⃣ ANÁLISIS Y RESULTADOS 📊** 7](#_Toc193881637)

[**5️⃣ RECOMENDACIONES Y PLAN DE ACCIÓN🏆** 16](#_Toc193881642)

[**6️⃣ CONCLUSIÓN 🎯** 17](#_Toc193881643)

**7️⃣ REPOSITORIO**  18

**8️⃣ PRÓXIMOS PASOS**🔜 19

[**9️⃣ REFERENCIAS TÉCNICAS ✅** 20](#_Toc193881644)

**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 🕒🐢

💾 **Infraestructura limitada:** La base de datos presenta sobrecargas.

**💼 Impacto en el Negocio**

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

⚠️ **Costos elevados:** La infraestructura actual no es eficiente.

**🏆 Soluciones Propuestas**

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

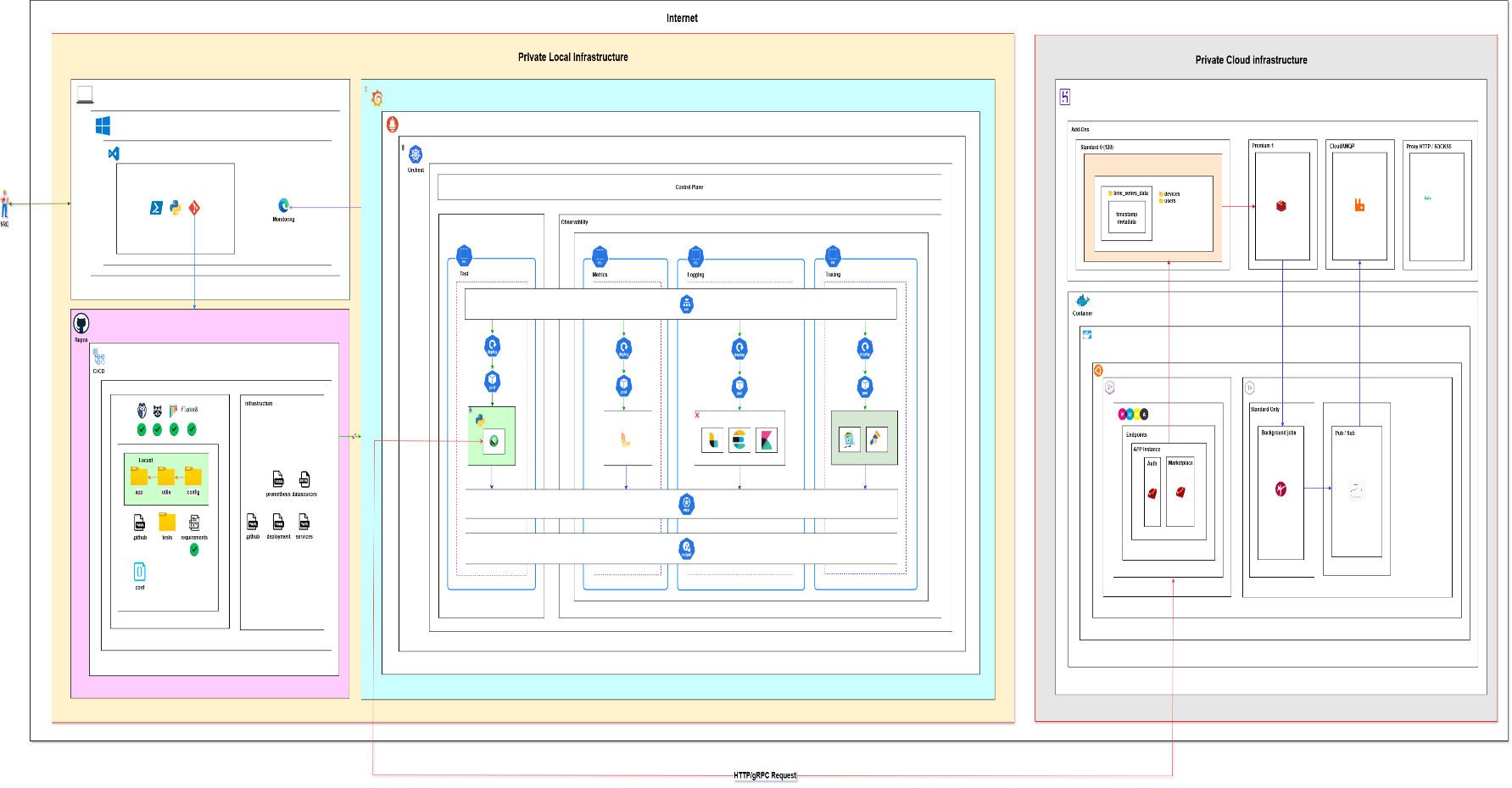
✅ Uso de CDN como **Cloudflare** para mejorar tiempos de respuesta 📌

**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.

**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.

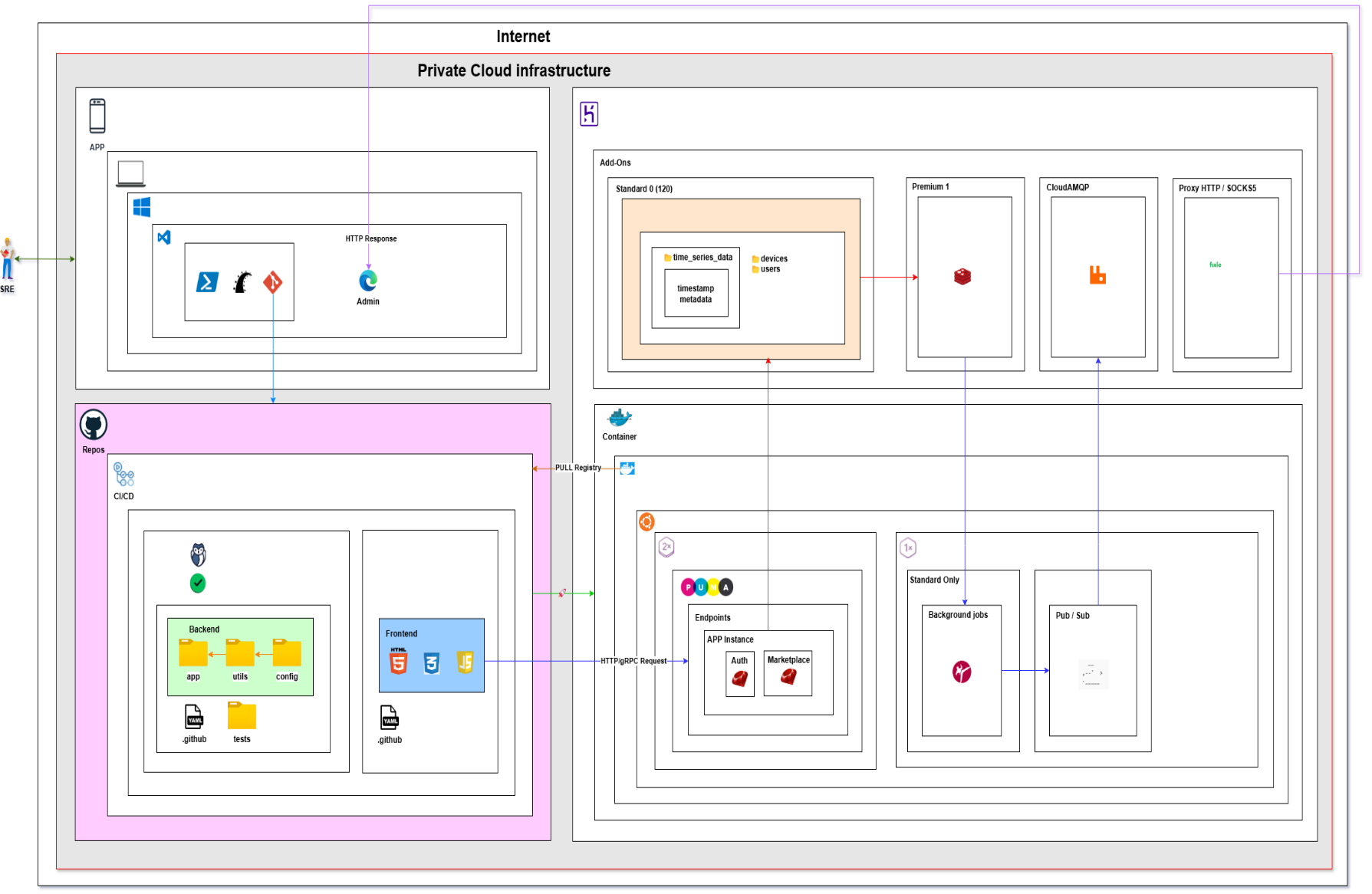
**ANÁLISIS Y RESULTADOS 📊**

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



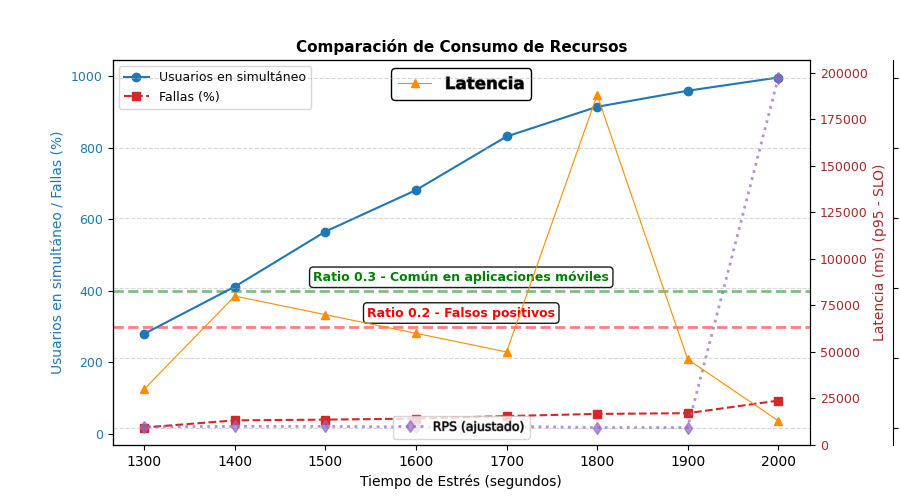
📌 **Capacidad actual:** La carga no alcanzó los 1000 usuarios esperados, y el sistema no superó el caching. Se requiere optimización.

**Infraestructura Actual en Heroku 🏗**

* 🔧 **Backend**: Desplegado en contenedores Docker ejecutando **Ruby on Rails** con **Puma** como servidor de aplicaciones 🐳
* **🐘 Base de datos:** PostgreSQL con presencia de **bloqueos en queries concurrentes**. Evaluar (Standard 2) para mayor capacidad de conexión🛢
* ⚡ **Caching**: Redis, con **expulsión de datos por falta de memoria**🔥
* 📨 **Pub/Sub**: Implementado con **RabbitMQ**, utilizado por **Sneakers** para el manejo de colas de eventos.
* 🐍 **Procesos en segundo plano**: Lotes (batch jobs) ejecutados bajo demanda mediante workers. Se utiliza **Sidekiq**.

**Análisis de Recursos y Costos 💰**

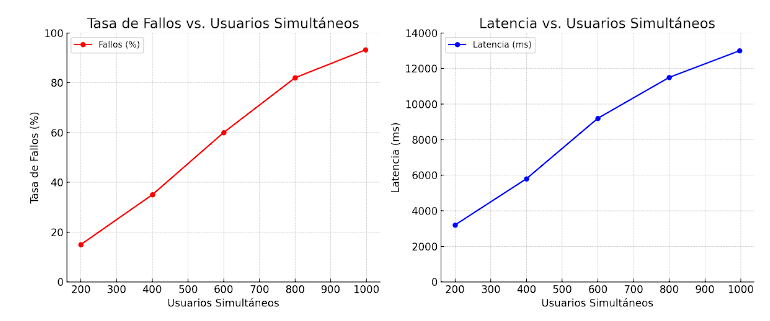
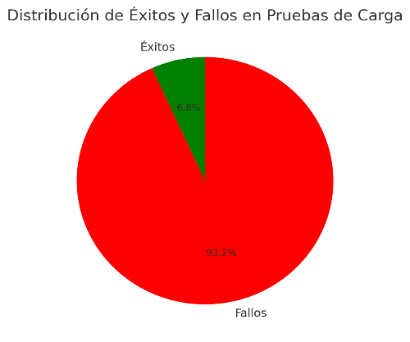
|  |  |  |
| --- | --- | --- |
| **Recurso** | **Estado Actual en Heroku** | **>= 1000 usuarios constantes** |
| 💾 **Dynos web (Puma)** | **X2** | **Optimizar x6 Performance M** 🔄 |
| 🛢 **PostgreSQL** | **Sin clustering** ❌ | **Escalar a mayor capacidad** 📈 |



📉 **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​)** |

* Permite **estimar cuántas conexiones activas** a la base de datos se requerirán durante un escenario de carga o uso real, en función de cómo la aplicación maneja las consultas.

⚙️ **Factor = 1** → Cada usuario genera **una consulta nueva por interacción** (no hay uso de caché, ni reuso de datos). Esto representa un escenario de **uso intensivo** y sin optimización.

⚙️ **Factor = 5** → Se estima que **solo 1 de cada 5 usuarios** genera una consulta al mismo tiempo, gracias a:

* Uso efectivo de **caching** (Redis).
* **Colas asincrónicas** para trabajo en segundo plano.
* **Procesamiento diferido** (Sidekiq, background jobs).
* **Pre-carga** de datos comunes (feeds, dashboards).

**🧪 Factor Aplicado**

Actualmente usan:

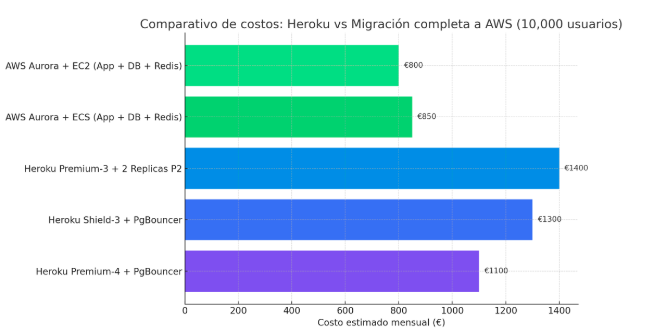
➡️ **Heroku PostgreSQL Standard 0** → Límite: **120 conexiones** activas simultáneas aproximadas, **no es suficiente.**

🔹 En pruebas con Locust, se simulan hasta **1000 usuarios simultáneos**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Escenario** | **Usuarios Simultáneos** | **Factor** | **Conexiones Necesarias** | **Plan** |
| **Sin caching / alta carga** | **1K** | **1** | **1K** | **Standard 0 - Heroku** |
| **Con buen caching** | **5** | **200** | **Standard 2 – Heroku (500)** |
| **10K** | **2K** | **Heroku Enterprise + PgBouncer** |
| **Sin caching / alta carga** | **1** | **10K** | **AWS** |

**Tabla de valores 📌**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Plataforma** | **Latencia 🚀** | **Escalabilidad (ms)** 🔄 | **Estimado (€ / mes) 💰** | **Uso 🔧** |
| **Premium-4** 🏗 | **Media (limitada por conexiones)** ❌ | | **1.1K** | **Fácil de administrar.** ✅ |
| **Shield-3 (Enterprise)** 🏗 | **Baja (Private Space)** ⚡ | **Alta (con costo elevado)** 📈 | **1.3K** | **Entorno dedicado, seguro y compliant (HIPAA/PCI). Costo elevado, manejo sencillo.** |
| **AWS Aurora Serverless v2 (32 ACUs) + ECS Spot Fargate (App + DB + Redis)** ☁️ | **Variable (optimizable con región / CDN)** | **Alta (autoscaling real)** 📈 | **850** | **Requiere gestión de infraestructura y observabilidad.** 🛠 |

➡️ **Las soluciones Heroku (1.2K u), necesitan PgBouncer para llegar a los 10K usuarios virtuales.**

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

➡️ **Se debe contar con un tiempo de gracias de 3 semanas (5 dias), calendario Colombia para validar la migración operativa.**

**RECOMENDACIONES Y PLAN DE ACCIÓN🏆**

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

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

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

**CONCLUSIÓN 🎯**

📢 El presente actualiza el 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 🚀

**REPOSITORIO**

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.

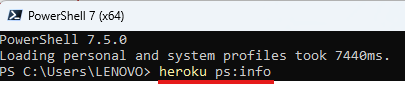
**PRÓXIMOS PASOS** 🔜

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Recurso** | **Pruebas pendientes en Heroku** | **Aplicado** | **Prueba de carga y estrés** | **Superado** |
| 🛢 **PostgreSQL** | 📌 **Configurar Heroku Enterprise con PgBouncer** |  |  |  |
| 💾 **AWS** | 📌 **Realizar la migración aceptada.** |  |  |  |

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

**REFERENCIAS TÉCNICAS ✅**

**🔧 Optimizar Dynos con PUMA**

* ****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 🎯

max\_requests = 1000

max\_requests\_jitter = 100

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

keepalive = 5

timeout = 30

* + **1 dyno bien configurado** puede manejar **300 usuarios**.
  + Para **1000 usuarios**, podrías necesitar **4 dynos**, reduciendo el costo en un **60% haciendo uso del auto escalado habilitado.**

**Habilitar logs y monitoreo**

* Utilizar herramientas como **Papertrail** o **LogDNA** para errores en tiempo real.
* Configurar un sistema de alertas para prevenir fallos críticos🔹

**🔥 Optimizar 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🔹
* Implementar **pooling de conexiones** (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.