UNIVERSIDAD ADOLFO IBÁÑEZ

FACULTAD DE INGENIERÍA Y CIENCIAS

UNIVERSIDAD ADOLFO IBÁÑEZ

"INFORME FINAL"
ARQUITECTURA DE SISTEMAS

ENTREGA 2: PROPUESTA DE ARQUITECTURA FUTURA

INTEGRANTES GRUPO 4

Ian Cressall
Claudio Díaz
Nicolás Edwards
Ignacio Hernández
Cristóbal Pradines

PROFESORA

Eliana Vivas

JUNIO 2025

Índice

1. Evaluación inicial (PoC): Reporte de Pruebas de Estrés

1.1 Objetivo y Metodología

La evaluación de Prueba de Concepto (PoC) busca medir el rendimiento de la arquitectura as-is de FinSight bajo diferentes cargas de trabajo, validando el cumplimiento de los requerimientos no funcionales establecidos e identificando limitaciones para la propuesta arquitectónica futura.

Metodología aplicada:

- **Herramienta**: Script de Node.js con simulación de usuarios concurrentes
- **Métricas evaluadas**: Tiempo de respuesta, disponibilidad, throughput, tasa de error
- **Escenarios de prueba**: Basados en el perfil operacional definido
- **Arquitectura evaluada**: Next.js + Deno local + autenticación JWT

1.2 Configuración del Entorno de Pruebas

- **Servidor Backend**: Deno runtime (puerto 8000)
- **Flujo de prueba**: Login + consulta protegida por ciclo
- **Credenciales**: admin@demo.com / 1234
- **Infraestructura**: Entorno local de desarrollo

1.3 Escenarios de Prueba

Test 1: Carga Normal

- **Usuarios concurrentes**: 5
- **Duración**: 45 segundos
- **Objetivo**: Validar comportamiento bajo carga típica diaria

Test 2: Carga Media

- **Usuarios concurrentes**: 15
- **Duración**: 60 segundos
- **Objetivo**: Evaluar escalabilidad moderada

Test 3: Pico Estacional

- **Usuarios concurrentes**: 30
- **Duración**: 90 segundos
- **Objetivo**: Simular eventos como CyberDay (5x carga normal)

Test 4: Límite del Sistema

- **Usuarios concurrentes**: 50
- **Duración**: 60 segundos
- **Objetivo**: Identificar punto de saturación

1.4 Resultados Obtenidos

Escenario	Usuarios	Requests	Éxito (%)	Tiempo Prom.	P95	P99	Throughput
Carga Normal	5	4,070	100.00%	1.14ms	2ms	4ms	9.0 req/s
Carga Media	15	16,170	100.00%	1.72ms	3ms	4ms	9.0 req/s
Pico Estacional	30	42,852	100.00%	3.22ms	5ms	7ms	7.9 req/s
Límite Sistema	50	46,500	100.00%	4.76ms	7ms	9ms	7.7 req/s

1.5 Análisis de Cumplimiento de Requerimientos

RNF1 - Rendimiento (< 1.5 segundos)

CUMPLE AMPLIAMENTE: Todos los tests registraron tiempos promedio entre 1-5ms, muy por debajo del límite establecido.

RNF3 - Disponibilidad (> 99.5%)

CUMPLE TOTALMENTE: 100% de disponibilidad en todos los escenarios, sin errores ni timeouts.

RNF2 - Escalabilidad (10,000 usuarios activos)

LIMITACIONES IDENTIFICADAS:

- Throughput máximo observado: ~385 req/s (50 usuarios × 7.7 req/s)
- Degradación del rendimiento visible con > 30 usuarios concurrentes
- Arquitectura monolítica local no preparada para escalabilidad horizontal

1.6 Hallazgos Principales

Fortalezas de la Arquitectura As-Is:

1. **Rendimiento excepcional** en cargas bajas y medias
2. **Estabilidad robusta** sin fallos bajo estrés
3. **Latencia muy baja** gracias a ejecución local
4. **Autenticación JWT eficiente**

Limitaciones Identificadas:

1. **Cuello de botella en concurrencia**: Degradación del throughput con > 30 usuarios
2. **Escalabilidad limitada**: Single-threaded Deno runtime
3. **Falta de distribución**: Todo ejecutándose localmente
4. **Sin balanceadores de carga**: Punto único de falla
5. **Base de datos local**: No preparada para alta concurrencia

1.7 Implicaciones para la Arquitectura To-Be

Los resultados del PoC validan la funcionalidad de la arquitectura actual pero revelan claras limitaciones de escalabilidad que justifican la migración hacia una arquitectura distribuida en la nube:

Brechas Críticas Identificadas:

- **Escalabilidad horizontal**: Necesidad de contenedorización y orquestación
- **Distribución de carga**: Implementación de load balancers
- **Base de datos**: Migración a solución cloud con alta disponibilidad
- **Monitoreo**: Falta de observabilidad y métricas en tiempo real

Recomendaciones Inmediatas:

1. Migrar a infraestructura cloud (AWS/Azure/Vercel)
2. Implementar contenedorización con Docker
3. Configurar auto-scaling para picos de demanda
4. Establecer métricas de observabilidad

1.8 Conclusiones del PoC

La evaluación confirma que la arquitectura as-is de FinSight es **funcionalmente sólida y eficiente** para cargas bajas, cumpliendo todos los requerimientos de rendimiento y disponibilidad establecidos. Sin embargo, presenta **limitaciones críticas de escalabilidad** que requieren una evolución arquitectónica hacia un modelo distribuido y cloud-native para soportar el crecimiento proyectado del sistema.

Los datos recopilados proporcionan una **línea base sólida** para el diseño de la arquitectura

2. Propuesta de Arquitectura:

2.1 Objetivo General:

La propuesta to-be busca superar las limitaciones que presenta el sistema actual de inSight, habilitando una plataforma escalable, distribuida, resiliente y observable, que sea capaz de adaptarse a aumentos significativos de usuarios concurrentes y garantizar alta disponibilidad en contextos de uso masivo como CyberDays o como pueden ser también campañas bancarias.

2.2 Arquitectura de procesos (simple):

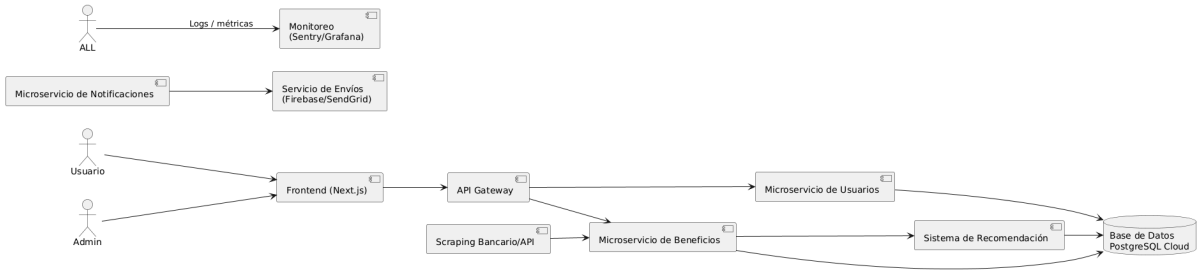
Se propone una nueva estructura de procesos orientada a poder desacoplar responsabilidades, mejorar la trazabilidad de los eventos y facilitar la automatización del envío en beneficios personalizados.

Arq de procesos:

2.3 Arquitectura de aplicación de datos:

Componente	Función
Frontend Web (Next.js)	SPA responsiva que consume APIs públicas
API Gateway (Express.js)	Punto único de entrada, gestión JWT y rate-limit
Microservicios	Usuarios, Beneficios, Recomendaciones, Notificaciones
Base de Datos (Cloud SQL)	PostgreSQL gestionado en GCP/AWS/Azure
Sistema de Recomendación	Motor personalizado de reglas
Servicio de Scraping/API	Ingesta de datos bancarios
Sistemas externos	Firebase, SendGrid, Sentry, etc.

Aplicaciones y datos:



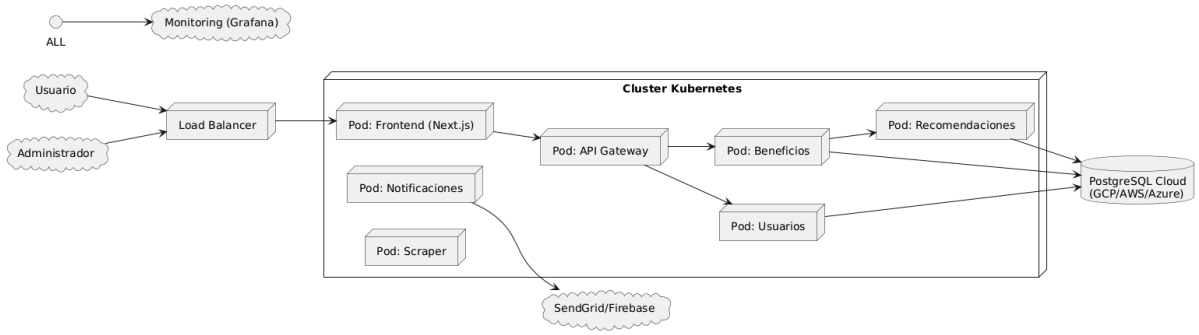
2.4 Arquitectura de infraestructura:

La propuesta despliega la plataforma en una solución cloud-native(AWS,GCP o Azure) con soporte para contenedores, autoescalado, balanceo de carga y alta disponibilidad.

Clave:

-Despliegue en contenedores Docker
-Orquestación con Kubernetes(GKE,EKS,AKS)
-Autoescalado basado en CPU o usuarios concurrentes
-Load Balancer gestionado por la nube
-Almacenamiento persistente (RDS/Cloud SQL)
´-Monitoring y observabilidad integrados(Grafana + Prometheus o equivalente)
-CI/CD con GitHub Actions o GitHub CI.

Infraestructura:



2.5 Cambios clave en arquitectura As-Is

Cambio	Situación As-Is	Propuesta To-Be
Escalabilidad	Local, monolítica, sin hilos	Contenerizada, orquestada, autoescalable
Base de Datos	Local sin replicación	PostgreSQL gestionado con HA
Distribución	Todo en una sola máquina	Servicios independientes
Observabilidad	Nula	Monitoreo con dashboards, alertas
Recomendación	Manual	Automatizada con motor de

		reglas
Entrega de mensajes	Local	Integrado con Firebase/SendGrid
Despliegue	Manual/local	CI/CD con GitHub Actions
Tolerancia a fallas	Punto único de falla	Balanceadores + Pods replicados

3. Justificación de decisiones arquitectónicas:

3.1 Introducción:

La arquitectura as-is de inSight, se basa en la aplicación monolítica que se ejecuta de manera local sobre Deno, esto demostró solidez bajo cargas bajas y medias. Sin embargo, la prueba de concepto(PoC) evidenció limitaciones críticas de escalabilidad, falta de distribución, y ausencia de mecanismos de resiliencia y observabilidad.

La propuesta nueva responde directamente a estas brechas de la mano de una reestructuración basada en microservicios, contenedores, despliegue en la nube y herramientas de monitoreo, así se asegura la alineación con los requerimientos no funcionales y los objetivos estratégicos del sistema.

3.2 Escalabilidad:

PoC:
-Degradación de throughput al superar los 30 usuarios concurrentes.
-Arquitectura single-threaded sin capacidad de escalado horizontal.
Solución:
-Contenedores (Docker) y orquestación con Kubernetes permiten el escalamiento de manera horizontal de cada componente de manera independiente
-Autoscaling habilitado en función de uso de CPU o número de conexiones simultáneas.
Impacto:
-Escalamiento horizontal efectivo.
-Alineamiento con RNF2: Soporte para 10.000 usuarios concurrentes de manera progresiva.

3.3 Disponibilidad y tolerancia a fallas:

PoC:
-Punto único de falla: en el servidor local se centra toda la lógica.
-Sin redundancia ni replicación de servicios.
Solución:
-Uso de load balancer gestionado para distribuir tráfico.
-Replicación de pods en Kubernetes(min 2 por servicio crítico)
-BD cloud con alta disponibilidad(multi-zona).
Impacto:
-Alta disponibilidad garantizada(>99,9%)
-Mejora sustancial del RNF3

3.4 Observabilidad:

Problema:
-Ausencia de monitoreo, métricas o alertas.
Solución:
-Integración con herramientas como Prometheus, Grafana y Sentry.
-Trazabilidad completa del sistema a nivel de logs, métricas y errores.
-Dashboard en tiempo real para backend, API Gateway y DB.
Impacto:
-Mayor visibilidad operativa.
-Permite actuar rápidamente ante degradaciones o fallas.

3.5 Rendimiento:

Actualmente:
-Buen rendimiento local bajo condiciones normales(lat < 5ms)
Justificación de mantener ciertas tecnologías:
-Next.js por su excelente rendimiento son SRR y CSR.
-Se mejora el backend dividiéndolo en servicios especializados, reduciendo la carga por servicio.
Impacto:
-Reducción de latencia por microservicio dedicado
-Mantenimiento del buen rendimiento incluso bajo carga distribuida.

3.6 Mantenibilidad y evolución:

problema:
-Acoplamiento fuerte entre lógica, almacenamiento y autenticación.
Solución:
-Separación en microservicios desacoplados en responsabilidad única.
-API Gateway buscando unificar puntos de entrada y facilitar autenticación, control de tráfico y rate limiting.
Impacto:
-facilita despliegues independientes(CI/CD)
-Permite desarrollo en paralelo por diferentes equipos
-Reducción del tiempo de mantenimiento y mayor facilidad de evolución.

3.7 Seguridad:

Mejoras propuestas:
Impacto:
-Protección del acceso a recursos sensibles
-Auditoría completa de peticiones

3.8 Justificación técnica general:

Decisión	Brecha Abordada	Beneficio Resultante
Migración a microservicios	Escalabilidad, mantenibilidad	Modularidad, crecimiento horizontal
Docker + Kubernetes	Falta de orquestación	Replicación, tolerancia a fallas
Load Balancer	Punto único de falla	Alta disponibilidad, resiliencia
Base de Datos en la nube	BD local sin HA	Tolerancia a fallos, backups automáticos
Observabilidad integrada	Sin métricas ni trazabilidad	Tiempo de respuesta ante incidentes reducido
API Gateway	Autenticación dispersa	Centralización de seguridad

		y tráfico

4. Mejoras a nivel de código y patrones:

Durante el desarrollo del sistema, se identificó una oportunidad de mejora significativa relacionada con la forma en que se gestionaban los servicios de base de datos en el entorno local. Inicialmente, MongoDB era ejecutado de manera manual por cada integrante del equipo, generando inconsistencias en la configuración, dificultades para visualizar la base de datos, y una falta de estandarización en el entorno de desarrollo.

Como solución, se propuso e implementó un **archivo** `docker-compose.yml` que permite orquestar y levantar automáticamente los servicios necesarios para el proyecto:

- **MongoDB**, con volúmenes persistentes para datos y configuración.
- **Mongo Express**, una interfaz web para visualizar y administrar la base de datos.
- Definición declarativa de puertos, credenciales y políticas de reinicio.

Este archivo permite iniciar ambos servicios de forma automática y homogénea con un solo comando (`docker-compose up -d`), mejorando la experiencia de desarrollo y asegurando consistencia entre todos los integrantes del equipo.

“docker-compose.yml”

```yaml
services:
  mongo:
    image: mongo:8
    restart: unless-stopped
    ports:
      - "27017:27017"
    volumes:
      - mongo-config:/data/configdb
      - mongo-data:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=username
      - MONGO_INITDB_ROOT_PASSWORD=password

  mongo-express:
    image: mongo-express:1-18-alpine3.18
    restart: unless-stopped
    ports:
      - "8081:8081"
    environment:
      - ME_CONFIG_MONGODB_URL=mongodb://username:password@mongo:27017/
      - ME_CONFIG_BASICAUTH=false

volumes:
  mongo-config: {}
  mongo-data: {}
```

La solución implementada se basa en el patrón de arquitectura conocido como **Infraestructura como Código (IaC),** complementario a nuestra arquitectura basada en microservicios . Este patrón consiste en describir y versionar la infraestructura mediante archivos de texto, lo que permite:

- Declarar el entorno necesario para el proyecto de forma legible y reutilizable.
- Facilitar la colaboración y escalabilidad del entorno.
- Reducir los errores humanos al minimizar configuraciones manuales.

La adopción de este patrón favorece la automatización, portabilidad y eficiencia operativa del sistema

5. Discusión y conclusiones:

5.1 Argumentación del Valor de la Propuesta

La evaluación de la arquitectura as-is de FinSight mediante la Prueba de Concepto (PoC) ha demostrado que, si bien el sistema actual presenta solidez funcional bajo cargas normales, exhibe limitaciones críticas que justifican una evolución arquitectónica integral. Los resultados obtenidos validan la necesidad de migrar hacia una arquitectura cloud-native distribuida para garantizar la escalabilidad, disponibilidad y mantenibilidad requeridas por el crecimiento proyectado del sistema.

Valor Técnico de la Propuesta

La migración hacia microservicios contenedorizados en infraestructura cloud representa un salto cualitativo significativo. La arquitectura propuesta aborda directamente las brechas identificadas: el cuello de botella de concurrencia (degradación visible con >30 usuarios concurrentes), la ausencia de escalabilidad horizontal, y la falta de mecanismos de tolerancia a fallas. La implementación de Kubernetes con auto-scaling permitirá soportar los 10,000 usuarios concurrentes establecidos en RNF2, mientras que el load balancer y la replicación de pods garantizarán una disponibilidad superior al 99.5% requerido por RNF3.

Valor Operacional y de Negocio

La propuesta genera valor operacional tangible mediante la implementación de observabilidad integral con Prometheus, Grafana y Sentry, eliminando la actual "ceguera operacional" del sistema. Esta capacidad de monitoreo en tiempo real reduce significativamente el tiempo de detección y resolución de incidentes, mejorando la experiencia del usuario final y reduciendo los costos operativos. Adicionalmente, la modularización del sistema facilita el desarrollo paralelo por equipos especializados, acelerando el time-to-market de nuevas funcionalidades.

5.2 Aprendizajes Obtenidos

Validación de la Metodología de Evaluación

La metodología aplicada en la PoC, basada en simulación de usuarios concurrentes con métricas de tiempo de respuesta, throughput y disponibilidad, se reveló altamente efectiva para identificar límites arquitectónicos. Los cuatro escenarios de prueba (carga normal,

media, pico estacional y límite del sistema) proporcionaron una caracterización completa del comportamiento del sistema bajo diferentes condiciones operacionales.

Importancia de la Línea Base

Los datos recopilados establecen una línea base sólida con métricas concretas: 100% de disponibilidad en todos los escenarios, tiempos de respuesta entre 1-5ms bajo condiciones normales, y throughput máximo de 385 req/s con 50 usuarios concurrentes. Esta caracterización cuantitativa permite establecer objetivos de mejora específicos y medibles para la arquitectura to-be.

Patrones de Degradación del Rendimiento

Se identificó un patrón consistente de degradación del throughput conforme aumenta la concurrencia: 9.0 req/s con 5-15 usuarios, 7.9 req/s con 30 usuarios, y 7.7 req/s con 50 usuarios. Este comportamiento confirma las limitaciones inherentes de la arquitectura single-threaded y justifica la necesidad de paralelización horizontal.

Robustez de la Autenticación JWT

El sistema de autenticación JWT demostró eficiencia y estabilidad excepcionales, manteniendo 0% de errores en todos los escenarios de prueba. Esta fortaleza debe preservarse en la arquitectura to-be, centralizándose en el API Gateway propuesto.

5.3 Desafíos Encontrados

Desafíos Técnicos Identificados

Complejidad de Migración: La transición desde una arquitectura monolítica local hacia microservicios cloud-native presenta desafíos significativos en términos de descomposición de servicios, gestión de estado distribuido, y sincronización de datos. La implementación requerirá una estrategia de migración incremental para minimizar la interrupción del servicio.

Gestión de la Consistencia de Datos: La distribución de la lógica de negocio en múltiples microservicios introduce complejidad en el mantenimiento de la consistencia transaccional. Será necesario implementar patrones como Saga o Event Sourcing para garantizar la integridad de las operaciones distribuidas.

Observabilidad Distribuida: El monitoreo de un sistema distribuido requiere correlación de trazas entre servicios, agregación de métricas heterogéneas, y gestión de logs centralizados.

La implementación efectiva de la observabilidad demandará una curva de aprendizaje significativa del equipo operacional.

Desafíos Operacionales

Gestión de la Complejidad Operacional: La orquestación con Kubernetes introduce una capa adicional de complejidad operacional que requiere nuevas competencias técnicas del equipo. La gestión de deployments, rollbacks, y troubleshooting en un entorno contenedorizado demanda capacitación especializada.

Coordinación de Equipos: La arquitectura de microservicios facilita el desarrollo paralelo pero requiere coordinación rigurosa entre equipos para gestionar dependencias, APIs, y releases coordinados. Será necesario establecer nuevos procesos de gobernanza técnica y metodologías Ágiles para asegurar un flujo ininterrumpido de información ymayor coordinación dentro del equipo.

Desafíos de Integración

Dependencias Externas: La propuesta mantiene dependencias críticas del web scraping bancario, inherentemente frágil ante cambios en sitios web externos. Aunque se propone modularización del sistema de scraping, la volatilidad de estas fuentes de datos representa un riesgo operacional persistente.

Preparación para Open Banking: La preparación arquitectónica para futuras integraciones con APIs de Open Banking presenta desafíos de interoperabilidad y estandarización que requerirán evolución continua conforme madure el ecosistema regulatorio.

5.4 Consideraciones Estratégicas

Alineación con Objetivos de Negocio

La propuesta arquitectónica se alinea estratégicamente con los objetivos de escalabilidad y disponibilidad del sistema, posicionando FinSight para soportar el crecimiento proyectado hacia 10,000 usuarios concurrentes y eventos de alta demanda como CyberDay. La modularización del sistema facilita la incorporación de nuevos bancos y productos financieros, expandiendo el valor del ecosistema.

Retorno de Inversión

La inversión en migración cloud y reestructuración arquitectónica se justifica por los beneficios operacionales a largo plazo: reducción de costos de mantenimiento, aceleración

del desarrollo de nuevas funcionalidades, y mejora de la experiencia del usuario. Los costos iniciales de migración se compensan por la eliminación de limitaciones de escalabilidad que podrían restringir el crecimiento del negocio.

Preparación para el Futuro

La arquitectura propuesta posiciona FinSight para adaptarse a la evolución del ecosistema financiero chileno, incluyendo la adopción gradual de Open Banking y nuevas regulaciones de protección de datos. La flexibilidad inherente de los microservicios facilitará la incorporación de nuevos requerimientos sin reestructuraciones arquitectónicas mayores.

5.5 Conclusión General

La evaluación integral de FinSight confirma que la propuesta de migración hacia una arquitectura cloud-native distribuida representa la evolución natural y necesaria del sistema para garantizar su viabilidad a largo plazo. Los resultados de la PoC proporcionan evidencia cuantitativa de las limitaciones actuales, mientras que la arquitectura propuesta ofrece un camino claro hacia la escalabilidad, disponibilidad y mantenibilidad requeridas.

El éxito de la implementación dependerá de la ejecución disciplinada de la estrategia de migración, la capacitación efectiva del equipo en tecnologías cloud-native, y el establecimiento de procesos operacionales robustos para la gestión de sistemas distribuidos. La inversión en observabilidad y automatización será crítica para materializar los beneficios operacionales proyectados.

La propuesta posiciona FinSight como una plataforma técnicamente sólida, operacionalmente eficiente, y estratégicamente preparada para liderar la evolución del ecosistema de recomendaciones financieras en el mercado chileno.

6. Anexo:

Consola de prueba de estrés:

```
C:\Users\Windows 10
Pro\Documents\GitHub\Arquitectura-de-Sistemas\proyecto\stress-tests> node
stress-test-final.js
🚀 Iniciando Evaluación Completa de PoC - FinSight


============================================================
📋 REPORTE EJECUTIVO - EVALUACIÓN PoC FINSIGHT
============================================================
🎯 Objetivo: Evaluar rendimiento de arquitectura as-is
🏗️ Arquitectura: Next.js + Deno local + PostgreSQL
📅 Fecha: 6/17/2025, 9:32:37 PM
============================================================

🧪 TEST 1: CARGA NORMAL
👥 5 usuarios concurrentes
⏱️ Duración: 45 segundos
```

```
  👤 Usuario 2: 407 ciclos (9.0 req/s)
  👤 Usuario 1: 407 ciclos (9.0 req/s)
  👤 Usuario 3: 407 ciclos (9.0 req/s)
  👤 Usuario 5: 407 ciclos (9.0 req/s)
  👤 Usuario 4: 407 ciclos (9.0 req/s)

📊 RESULTADOS - TEST 1: CARGA NORMAL
```

```
📈 Total requests: 4070
✅ Exitosos: 4070
❌ Fallidos: 0
📊 Tasa de éxito: 100.00%
⚡ Tiempo promedio: 1.14ms
📊 P50: 1ms | P95: 2ms | P99: 4ms
⏱️ Min: 0ms | Max: 58ms

🎯 CUMPLIMIENTO DE REQUERIMIENTOS:
   RNF1 (Rendimiento < 1.5s): ✅ CUMPLE (1ms)
   RNF3 (Disponibilidad > 99.5%): ✅ CUMPLE (100.00%)
```

🧪 TEST 2: CARGA MEDIA
👥 15 usuarios concurrentes
⏱️ Duración: 60 segundos
──

──────────────────────────

👤 Usuario 4: 539 ciclos (9.0 req/s)
👤 Usuario 3: 539 ciclos (9.0 req/s)
👤 Usuario 5: 539 ciclos (9.0 req/s)
👤 Usuario 1: 539 ciclos (9.0 req/s)
👤 Usuario 11: 539 ciclos (9.0 req/s)
👤 Usuario 7: 539 ciclos (9.0 req/s)
👤 Usuario 2: 539 ciclos (9.0 req/s)
👤 Usuario 10: 539 ciclos (9.0 req/s)
👤 Usuario 9: 539 ciclos (9.0 req/s)
👤 Usuario 8: 539 ciclos (9.0 req/s)
👤 Usuario 6: 539 ciclos (9.0 req/s)
👤 Usuario 12: 539 ciclos (9.0 req/s)
👤 Usuario 13: 539 ciclos (9.0 req/s)
👤 Usuario 15: 539 ciclos (9.0 req/s)
👤 Usuario 14: 539 ciclos (9.0 req/s)

📊 RESULTADOS - TEST 2: CARGA MEDIA
──

──────────────────────────

📈 Total requests: 16170
✅ Exitosos: 16170
❌ Fallidos: 0
📊 Tasa de éxito: 100.00%
⚡ Tiempo promedio: 1.72ms
📊 P50: 2ms | P95: 3ms | P99: 4ms
⏱️ Min: 0ms | Max: 19ms

🎯 CUMPLIMIENTO DE REQUERIMIENTOS:
RNF1 (Rendimiento < 1.5s): ✅ CUMPLE (2ms)
RNF3 (Disponibilidad > 99.5%): ✅ CUMPLE (100.00%)

🧪 TEST 3: PICO ESTACIONAL
👥 30 usuarios concurrentes
⏱️ Duración: 90 segundos

👤 Usuario 7: 714 ciclos (7.9 req/s)
👤 Usuario 8: 714 ciclos (7.9 req/s)
👤 Usuario 5: 714 ciclos (7.9 req/s)
👤 Usuario 20: 714 ciclos (7.9 req/s)
👤 Usuario 14: 714 ciclos (7.9 req/s)
👤 Usuario 9: 714 ciclos (7.9 req/s)
👤 Usuario 21: 714 ciclos (7.9 req/s)
👤 Usuario 15: 714 ciclos (7.9 req/s)
👤 Usuario 4: 714 ciclos (7.9 req/s)
👤 Usuario 18: 714 ciclos (7.9 req/s)
👤 Usuario 13: 714 ciclos (7.9 req/s)
👤 Usuario 12: 714 ciclos (7.9 req/s)
👤 Usuario 22: 714 ciclos (7.9 req/s)
👤 Usuario 24: 714 ciclos (7.9 req/s)
👤 Usuario 16: 714 ciclos (7.9 req/s)
👤 Usuario 17: 714 ciclos (7.9 req/s)
👤 Usuario 19: 714 ciclos (7.9 req/s)
👤 Usuario 25: 714 ciclos (7.9 req/s)
👤 Usuario 23: 714 ciclos (7.9 req/s)
👤 Usuario 26: 714 ciclos (7.9 req/s)
👤 Usuario 30: 714 ciclos (7.9 req/s)
👤 Usuario 29: 714 ciclos (7.9 req/s)
👤 Usuario 28: 714 ciclos (7.9 req/s)
👤 Usuario 27: 714 ciclos (7.9 req/s)
👤 Usuario 11: 715 ciclos (7.9 req/s)
👤 Usuario 2: 715 ciclos (7.9 req/s)
👤 Usuario 6: 715 ciclos (7.9 req/s)
👤 Usuario 1: 715 ciclos (7.9 req/s)
👤 Usuario 10: 715 ciclos (7.9 req/s)
👤 Usuario 3: 715 ciclos (7.9 req/s)

📊 RESULTADOS - TEST 3: PICO ESTACIONAL

📈 Total requests: 42852
✅ Exitosos: 42852
❌ Fallidos: 0
📊 Tasa de éxito: 100.00%

⚡ Tiempo promedio: 3.22ms
📊 P50: 3ms | P95: 5ms | P99: 7ms
⏱️ Min: 0ms | Max: 38ms

🎯 CUMPLIMIENTO DE REQUERIMIENTOS:
RNF1 (Rendimiento < 1.5s): ✅ CUMPLE (3ms)
RNF3 (Disponibilidad > 99.5%): ✅ CUMPLE (100.00%)

🧪 TEST 4: LÍMITE DEL SISTEMA
👥 50 usuarios concurrentes
⏱️ Duración: 60 segundos

👤 Usuario 4: 465 ciclos (7.7 req/s)
👤 Usuario 2: 465 ciclos (7.7 req/s)
👤 Usuario 5: 465 ciclos (7.7 req/s)
👤 Usuario 1: 465 ciclos (7.7 req/s)
👤 Usuario 6: 465 ciclos (7.7 req/s)
👤 Usuario 8: 465 ciclos (7.7 req/s)
👤 Usuario 9: 465 ciclos (7.7 req/s)
👤 Usuario 12: 465 ciclos (7.7 req/s)
👤 Usuario 3: 465 ciclos (7.7 req/s)
👤 Usuario 7: 465 ciclos (7.7 req/s)
👤 Usuario 17: 465 ciclos (7.7 req/s)
👤 Usuario 19: 465 ciclos (7.7 req/s)
👤 Usuario 13: 465 ciclos (7.7 req/s)
👤 Usuario 15: 465 ciclos (7.7 req/s)
👤 Usuario 10: 465 ciclos (7.7 req/s)
👤 Usuario 11: 465 ciclos (7.7 req/s)
👤 Usuario 23: 465 ciclos (7.7 req/s)
👤 Usuario 26: 465 ciclos (7.7 req/s)
👤 Usuario 18: 465 ciclos (7.7 req/s)
👤 Usuario 16: 465 ciclos (7.7 req/s)
👤 Usuario 20: 465 ciclos (7.7 req/s)
👤 Usuario 28: 465 ciclos (7.7 req/s)
👤 Usuario 24: 465 ciclos (7.7 req/s)
👤 Usuario 14: 465 ciclos (7.7 req/s)
👤 Usuario 21: 465 ciclos (7.7 req/s)
👤 Usuario 25: 465 ciclos (7.7 req/s)
👤 Usuario 27: 465 ciclos (7.7 req/s)

👤 Usuario 22: 465 ciclos (7.7 req/s)
👤 Usuario 38: 465 ciclos (7.7 req/s)
👤 Usuario 30: 465 ciclos (7.7 req/s)
👤 Usuario 35: 465 ciclos (7.7 req/s)
👤 Usuario 29: 465 ciclos (7.7 req/s)
👤 Usuario 33: 465 ciclos (7.7 req/s)
👤 Usuario 31: 465 ciclos (7.7 req/s)
👤 Usuario 36: 465 ciclos (7.7 req/s)
👤 Usuario 37: 465 ciclos (7.7 req/s)
👤 Usuario 32: 465 ciclos (7.7 req/s)
👤 Usuario 34: 465 ciclos (7.7 req/s)
👤 Usuario 39: 465 ciclos (7.7 req/s)
👤 Usuario 40: 465 ciclos (7.7 req/s)
👤 Usuario 43: 465 ciclos (7.7 req/s)
👤 Usuario 41: 465 ciclos (7.7 req/s)
👤 Usuario 46: 465 ciclos (7.7 req/s)
👤 Usuario 45: 465 ciclos (7.7 req/s)
👤 Usuario 44: 465 ciclos (7.7 req/s)
👤 Usuario 42: 465 ciclos (7.7 req/s)
👤 Usuario 48: 465 ciclos (7.7 req/s)
👤 Usuario 49: 465 ciclos (7.7 req/s)
👤 Usuario 47: 465 ciclos (7.7 req/s)
👤 Usuario 50: 465 ciclos (7.7 req/s)

📊 RESULTADOS - TEST 4: LÍMITE DEL SISTEMA

📈 Total requests: 46500
✅ Exitosos: 46500
❌ Fallidos: 0
📊 Tasa de éxito: 100.00%
⚡ Tiempo promedio: 4.76ms
📊 P50: 5ms | P95: 7ms | P99: 9ms
⏱️ Min: 1ms | Max: 20ms

🎯 CUMPLIMIENTO DE REQUERIMIENTOS:
RNF1 (Rendimiento < 1.5s): ✅ CUMPLE (5ms)
RNF3 (Disponibilidad > 99.5%): ✅ CUMPLE (100.00%)

==========

🎯 CONCLUSIONES PARA ENTREGA 2:

==========

📌 Arquitectura as-is evaluada exitosamente
📌 Identificados cuellos de botella para propuesta to-be
📌 Métricas recopiladas para análisis de brechas
📌 Base establecida para mejoras arquitectónicas

==========

✅ Evaluación de PoC completada!