UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CARRERA DE SOFTWARE

TRABAJO DE INTEGRACIÓN CURRICULAR

TEMA:

"BENCHMARKING DE FRAMEWORKS DE DESARROLLO, PARA LA CREACIÓN DE APIS DE ALTA DISPONIBILIDAD MEDIANTE MÉTRICAS DE RENDIMIENTO Y ESCALABILIDAD"

Trabajo de titulación previo a la obtención del título de ingeniero en software

Línea de investigación: Desarrollo, aplicación de software y cyber security (seguridad cibernética)

AUTOR:

Sergio Edwin Pazmiño Torres

DIRECTOR:

MSc. Diego Javier Trejo España

Ibarra – Ecuador

2026

UNIVERSIDAD TÉCNICA DEL NORTE
DIRECCIÓN DE BIBLIOTECA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO	
APELLIDOS Y NOMBRES:	Pazmiño Torres Sergio Edwin

DATOS DE LA OBRA	
TÍTULO:	Benchmarking de frameworks de desarrollo, para la creación de APIs de alta disponibilidad mediante métricas de rendimiento y escalabilidad
AUTOR (ES):	Pazmiño Torres Sergio Edwin
FECHA:	20/10/2026
SOLO PARA TRABAJOS DE GRADO	
PROGRAMA:	■ **PREGRADO** ☐ **POSGRADO**
TITULO POR EL QUE OPTA:	Ingeniero en Software
DIRECTOR /ASESOR:	MSc. Diego Javier Trejo España

2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es (son) el (los) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 20 días del mes de octubre de 2026

EL AUTOR:

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
SERGIO EDWIN
PAZMINO TORRES

.....

Nombre: Pazmiño Torres Sergio Edwin

CERTIFICACIÓN DEL DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR

Ibarra, 20 de octubre del 2026

MSc. Diego Javier Trejo España

DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR

CERTIFICA:

Haber revisado el presente informe final del trabajo de Integración Curricular, el mismo que se ajusta a las normas vigentes de la Universidad Técnica del Norte; en consecuencia, autorizo su presentación para los fines legales pertinentes.



MSc. Diego Javier Trejo España
Director de Trabajo de Grado

ÍNDICE DE CONTENIDOS

CAPITULO I: INTRODUCCIÓN 1
1.1 Problema de investigación. 1
1.2 Justificación 2
1.3 Objetivos 3
1.3.1 Objetivo General 3
1.3.2 Objetivos Específicos 3
1.4 Hipótesis o preguntas de investigación. 4
CAPÍTULO II: MARCO TEÓRICO 5
2.1 Benchmarking 5
2.2 Lenguajes de programación 5
2.2.1 Go 5
2.2.2 JavaScript 6
2.2.3 Elixir 6
2.2.4 Modelos de concurrencia 7
2.3 API y Microservicio 10
2.3.1 API 10
2.3.2 Microservicio 11
2.4 Atributos de Calidad del Sistema 11
2.4.1 Rendimiento 11
2.4.2 Escalabilidad 11
2.4.3 Disponibilidad 12
2.4.4 Concurrencia 12
2.5 Infraestructura en la nube 12
2.6 Trabajos relacionados 13
CAPÍTULO III: MATERIALES Y MÉTODOS 17
3.1 Enfoque y tipo de investigación 17

3.2 Diseño de la investigación: Materiales, equipos y software 17
3.3 Métodos, técnicas e instrumentos. 19
3.3.1 Unidades experimentales y tamaño del experimento 19
3.3.2 Técnicas e instrumentos de recolección de datos 19
3.3.3 Matriz de operacionalización de variables. 20
3.4 Procedimiento y análisis de datos. 23
3.4.1 Procedimiento experimental 23
3.4.2 Análisis de datos 25
3.4.3 Aspectos éticos 26
CAPÍTULO IV: RESULTADOS Y ANÁLISIS 26
4.1 Arquitectura Monolítica 26
4.1.1 Tiempo medio de respuesta 26
4.1.2 Adecuación del tiempo de respuesta 29
4.1.3 Rendimiento medio 31
4.1.4 Utilización media del procesador 33
4.1.5 Utilización media de memoria 35
4.1.6 Utilización del ancho de banda 37
4.1.7 Capacidad de procesamiento de transacciones 39
4.1.8 Capacidad de acceso de usuarios 42
4.1.9 Adecuación al incremento del acceso de usuarios 44
4.2 Arquitectura de Microservicios 46
4.2.1 Tiempo medio de respuesta 46
4.2.2 Adecuación del tiempo de respuesta 47
4.2.3 Rendimiento medio 49
4.2.4 Utilización media del procesador 50
4.2.5 Utilización media de memoria 51
4.2.6 Utilización del ancho de banda 52

4.2.7 Capacidad de procesamiento de transacciones 53
4.2.8 Capacidad de acceso de usuarios 54
4.2.9 Adecuación al incremento del acceso de usuarios 55
4.3 Análisis comparativo global 56
4.3.1 Errores encontrados 56
4.3.2 Entorno Monolítico 57
4.3.3 Entorno de Microservicios 58
4.3.4 Síntesis Comparativa 58
4.4 Discusión 59
CONCLUSIONES 61
RECOMENDACIONES 62
REFERENCIAS BIBLIOGRÁFICAS 63
ANEXOS 67

ÍNDICE DE FIGURAS

Figura 1 Árbol de Problemas 2
Figura 2 Tiempo medio de respuesta - Usuario 27
Figura 3 Tiempo medio de respuesta - Factura 29
Figura 4 Adecuación del tiempo de respuesta - Usuario 30
Figura 5 Adecuación del tiempo de respuesta - Factura 31
Figura 6 Rendimiento medio - Usuario 32
Figura 7 Rendimiento medio - Factura 33
Figura 8 Utilización media del procesador - Usuario 34
Figura 9 Utilización media del procesador - Factura 35
Figura 10 Utilización media de memoria - Usuario 36
Figura 11 Utilización media de memoria - Factura 37
Figura 12 Utilización del ancho de banda - Usuario 38
Figura 13 Utilización del ancho de banda - Factura 39
Figura 14 Capacidad de procesamiento de transacciones - Usuario 40
Figura 15 Capacidad de procesamiento de transacciones - Factura 42
Figura 16 Capacidad de acceso de usuarios - Usuario 43
Figura 17 Capacidad de acceso de usuarios - Factura 44
Figura 18 Adecuación al incremento del acceso de usuarios - Usuario 45
Figura 19 Adecuación al incremento del acceso de usuarios - Factura 46
Figura 20 Tiempo medio de respuesta en Microservicios 47
Figura 21 Adecuación del tiempo de respuesta en Microservicios 49
Figura 22 Rendimiento medio en Microservicios 50
Figura 23 Utilización media del procesador en Microservicios 51
Figura 24 Utilización media de memoria en Microservicios 52
Figura 25 Utilización del ancho de banda en Microservicios 53
Figura 26 Capacidad de procesamiento de transacciones en Microservicios 54
Figura 27 Capacidad de acceso de usuarios en Microservicios 55
Figura 28 Adecuación al incremento del acceso de usuarios en Microservicios 56
Figura 29 Panel de Render 75

ÍNDICE DE TABLAS

Tabla 1 Materiales, equipos y software empleados en la investigación. 18
Tabla 2 Operacionalización de variable independiente 20
Tabla 3 Operacionalización de variables dependientes[34]..... 21
Tabla 4 Operacionalización de variables controladas..... 23
Tabla 5 Configuración de Pruebas de Carga y Estrés..... 24
Tabla 6 Tiempo medio de respuesta - Usuario 27
Tabla 7 Tiempo medio de respuesta - Factura..... 28
Tabla 8 Adecuación del tiempo de respuesta - Usuario..... 29
Tabla 9 Adecuación del tiempo de respuesta - Factura 31
Tabla 10 Rendimiento medio - Usuario..... 32
Tabla 11 Rendimiento medio - Factura 32
Tabla 12 Utilización media del procesador - Usuario 33
Tabla 13 Utilización media del procesador - Factura..... 34
Tabla 14 Utilización media de memoria - Usuario..... 35
Tabla 15 Utilización media de memoria - Factura..... 36
Tabla 16 Utilización del ancho de banda - Usuario..... 37
Tabla 17 Utilización del ancho de banda - Factura..... 38
Tabla 18 Capacidad de procesamiento de transacciones - Usuario 39
Tabla 19 Capacidad de procesamiento de transacciones - Factura..... 41
Tabla 20 Capacidad de acceso de usuarios - Usuario 42
Tabla 21 Capacidad de acceso de usuarios - Factura..... 43
Tabla 22 Adecuación al incremento del acceso de usuarios - Usuario..... 44
Tabla 23 Adecuación al incremento del acceso de usuarios - Factura 45
Tabla 24 Tiempo medio de respuesta en Microservicios..... 47
Tabla 25 Adecuación del tiempo de respuesta en Microservicios..... 48
Tabla 26 Rendimiento medio en Microservicios..... 49
Tabla 27 Utilización media del procesador en Microservicios..... 50
Tabla 28 Utilización media de memoria en Microservicios 51
Tabla 29 Utilización del ancho de banda en Microservicios 52
Tabla 30 Capacidad de procesamiento de transacciones en Microservicios 53
Tabla 31 Capacidad de acceso de usuarios en Microservicios 55
Tabla 32 Adecuación al incremento del acceso de usuarios en Microservicios 55

RESUMEN

La presente investigación fundamenta su desarrollo en el benchmarking y la experimentación comparativa, teniendo como aspectos clave, el rendimiento y la escalabilidad de APIs. Los servicios fueron implementados en tres frameworks: Express (Node.js), Gin (Go) y Phoenix (Elixir). El objetivo consistió en evaluar la eficiencia y alta disponibilidad de estas tecnologías mediante el estándar de calidad ISO/IEC 25023 y pruebas de carga y estrés. Para ello, se siguió una metodología experimental que abarcó la codificación de APIs con un CRUD simple, bajo arquitecturas monolíticas y de microservicios, sometiéndolos a pruebas de carga y estrés.

Los resultados revelaron que Phoenix aventaja en velocidad de procesamiento y concurrencia en transacciones simples; Gin demuestra una óptima eficiencia en el uso de recursos, CPU y memoria RAM, con una fiabilidad aproximada al 0% de errores, aunque con un menor soporte de usuarios simultáneos en el monolito; y Express, a pesar de tener el peor rendimiento general, sobresale por su resiliencia al sostener un mayor volumen de usuarios bajo persistencias complejas. En los microservicios, la incorporación de un API Gateway homogeneizó el rendimiento de las tres tecnologías, actuando como un cuello de botella de la infraestructura. Los resultados mostraron en que aspectos destacan los frameworks y que existen diferentes factores que pueden influir en su rendimiento.

Palabras clave: Benchmarking, ISO/IEC 25023, Microservicio, API, Rendimiento de framework.

ABSTRACT

This research is based on benchmarking and comparative experimentation, focusing on API performance and scalability. The services were implemented using three frameworks: Express (Node.js), Gin (Go), and Phoenix (Elixir). The objective was to evaluate the efficiency and high availability of these technologies through the ISO/IEC 25023 quality standard and load and stress testing. To achieve this, an experimental methodology was followed, involving the development of APIs with simple CRUD functionality under both monolithic and microservices architectures, which were subsequently subjected to load and stress tests.

The results revealed that Phoenix outperformed the other frameworks in processing speed and concurrency for simple transactions. Gin demonstrated optimal efficiency in CPU and RAM resource utilization, with reliability close to a 0% error rate, although supporting fewer simultaneous users in the monolithic architecture. Express, despite exhibiting the lowest overall performance, stood out for its resilience by sustaining a higher volume of users under complex persistence operations. In the microservices architecture, the incorporation of an API Gateway homogenized the performance of the three technologies, acting as an infrastructure bottleneck. The findings highlighted the strengths of each framework and demonstrated that multiple factors can influence their overall performance.

Keywords: Benchmarking, ISO/IEC 25023, Microservices, API, Framework performance.

LISTA DE SIGLAS

API. Interfaz de Programación de Aplicaciones (*Application Programming Interface*).

CPU. Unidad Central de Procesamiento (*Central Processing Unit*).

CRUD. Crear, Leer, Actualizar y Eliminar (*Create, Read, Update, Delete*).

E/S. Entrada y Salida.

HTTP. Protocolo de Transferencia de Hipertexto (*Hypertext Transfer Protocol*).

IEC. Comisión Electrotécnica Internacional (*International Electrotechnical Commission*).

ISO. Organización Internacional de Normalización (*International Organization for Standardization*).

JSON. Notación de Objetos de JavaScript (*JavaScript Object Notation*).

RAM. Memoria de Acceso Aleatorio (*Random Access Memory*).

RPS. Peticiones por Segundo (*Requests per Second*).

VU. Usuario Virtual (*Virtual User*).

CAPITULO I: INTRODUCCIÓN

1.1 Problema de investigación.

En la actualidad, las Interfaces de Programación de Aplicaciones (APIs) y microservicios son componentes esenciales para organizaciones e instituciones. Los sistemas multiusuario, transacciones que se ejecutan concurrentemente, presiones de crecimiento y el incremento del volumen de actividades, dan como resultado una creciente demanda de escalabilidad y un manejo eficiente de la concurrencia [1], [2]. Debido a implementaciones inconsistentes, las APIs pueden fallar al gestionar aumentos de carga de trabajo o picos de tráfico, un problema que se agrava ante la rápida adopción de arquitecturas de microservicios y la necesidad de transacciones en tiempo real [3]. Este problema se intensifica si tenemos en cuenta la selección de arquitecturas y tecnologías [4].

El desafío principal reside en la influencia de los frameworks en el rendimiento ante la alta concurrencia. Existe una incertidumbre significativa sobre cómo la selección del framework impacta el desempeño, agravada por la falta de un estudio comparativo estandarizado, imparcial y actualizado que analice diversos ecosistemas bajo una misma infraestructura. Esta carencia conduce a decisiones menos eficientes, posibles sobrecostos en infraestructura y otras consecuencias negativas.

Este problema puede conllevar a riesgos operacionales y estratégicos, en donde se ve afectado el usuario y la propia empresa, como se puede observar en la **Figura 1**.

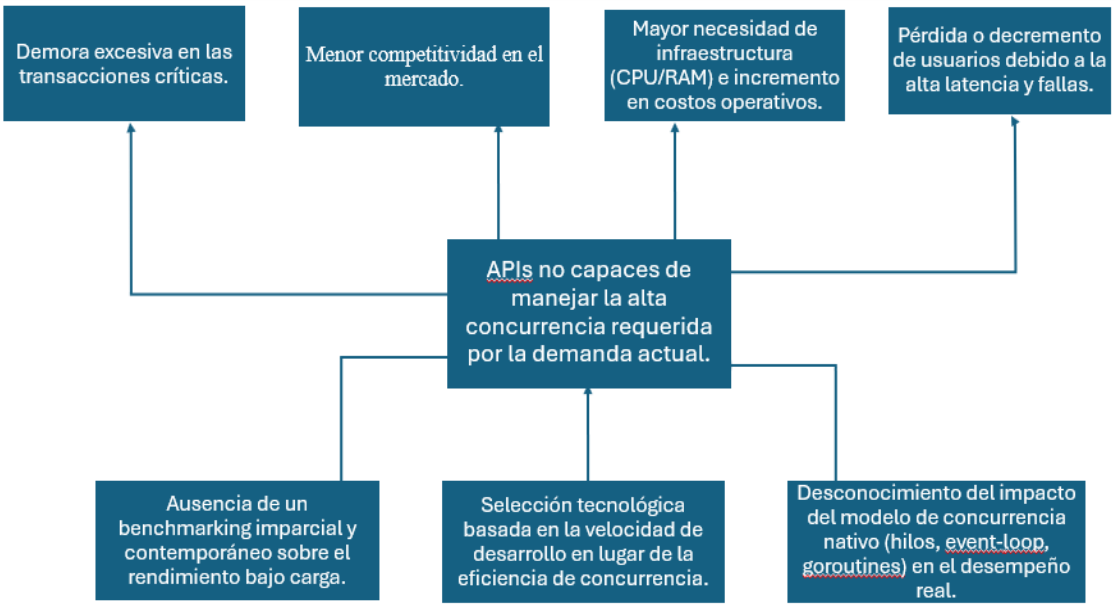


Figura 1 Árbol de Problemas

1.2 Justificación

El acelerado crecimiento de la globalización tecnológica está dando como resultado que las aplicaciones tengan la capacidad de manejar altas cargas de trabajo, destacando el manejo de la alta concurrencia. Según [2], la ejecución concurrente de transacciones puede generar interferencias, errores y, en consecuencia, problemas para los usuarios y la empresa. Ante este problema, la motivación central de esta investigación surge de la brecha de conocimiento existente en la comunidad de ingeniería, debido a que carece de un estudio comparativo imparcial y contemporáneo que enfrente a varios modelos de concurrencia nativos de los tres principales ecosistemas de backend (Go, Elixir, y JavaScript) bajo escenarios idénticos de carga masiva.

La presente investigación proporcionará datos empíricos sobre el rendimiento, facilitando la toma de decisiones informadas para arquitectos y desarrolladores al momento de seleccionar un lenguaje de programación para el desarrollo de APIs y microservicios. Esto permitirá minimizar el riesgo de sobredimensionamiento, asegurar la robustez del sistema desde la fase de diseño y optimizar el uso de recursos. Los beneficiarios de este estudio abarcan desde emprendedores hasta empresas establecidas que diseñan o refactorizan sus arquitecturas de microservicios, permitiendo una mayor escalabilidad y reduciendo la inversión en hardware. Además, la investigación beneficiará a la comunidad de desarrollo, al proporcionar información valiosa sobre el rendimiento de los frameworks.

Esta investigación se alinea con el Objetivo de Desarrollo Sostenible (ODS) número nueve, que busca construir infraestructuras resilientes y fomentar la industrialización inclusiva y sostenible, así como la innovación [5]. Adicionalmente, el estudio se vincula con el Objetivo 7 del Plan de Creación de Oportunidades 2021-2025, el cual enfatiza en impulsar la transformación digital y la innovación, garantizando la calidad de los servicios digitales [6].

Justificación académica: La preparación académica de calidad es crucial para que los estudiantes egresados puedan acceder al mercado laboral. Según [7], los principales problemas para los egresados son la falta de experiencia, la falta de conocimientos en nuevas tecnologías, herramientas y software, y falta de oportunidades. Esta investigación, al aportar conocimiento teórico y práctico, permitirá a los interesados comprender el rendimiento comparativo de los lenguajes de programación en el manejo de transacciones de alta concurrencia.

Justificación económica: En la actualidad, la elección de la tecnología adecuada es fundamental para el crecimiento sostenible de una empresa. Posponer esta decisión, según [8], genera deuda técnica, que se traduce en costos asociados a la ejecución y soporte de tecnologías imperfectas, aumentando la complejidad de los sistemas y dificultando su mantenimiento. Al identificar el framework más eficiente, la investigación contribuirá a la reducción de gastos innecesarios, a minimizar la necesidad de recursos de infraestructura y combatir la deuda técnica, proporcionando un manejo óptimo de la alta concurrencia y escalabilidad, lo cual se traduce en ahorro de costos y mayor eficiencia.

1.3 Objetivos

1.3.1 Objetivo General

Comparar tres frameworks de desarrollo, para la creación de APIs de alta disponibilidad mediante métricas de rendimiento y escalabilidad.

1.3.2 Objetivos Específicos

- Elaborar un marco teórico mediante la revisión de literatura académica y técnica, enfocada en comparaciones entre frameworks de desarrollo de APIs.
- Implementar el benchmarking mediante el despliegue de un entorno de infraestructura estandarizado en la nube, que incluye el desarrollo de tres APIs idénticas y la aplicación de pruebas de carga masiva para medir el rendimiento y

la escalabilidad de los frameworks de concurrencia nativos de Go, Elixir y JavaScript.

- Analizar los resultados obtenidos del benchmarking, utilizando métricas basadas en la ISO 25023, para identificar y documentar las ventajas, desventajas y el framework óptimo para el desarrollo de APIs de alta concurrencia.

1.4 Hipótesis o preguntas de investigación.

Las preguntas que guían esta investigación están enfocadas enteramente en el rendimiento de los frameworks, y son las siguientes:

- ¿Cuál framework ofrece la mayor disponibilidad para el desarrollo de APIs de alta concurrencia?
- ¿Cuál es el framework que mejor maneja la alta concurrencia con un enfoque maestro-detalle?
- ¿Cómo impacta el uso de un API Gateway en los microservicios?

La hipótesis planteada, la cual va a ser validad o refutada una vez finalizada la investigación, es la siguiente:

- Existen diferencias significativas en las métricas de rendimiento y escalabilidad entre las soluciones desarrolladas con Go, Elixir y JavaScript, evaluadas mediante la norma ISO/IEC 25023.

CAPÍTULO II: MARCO TEÓRICO

2.1 Benchmarking

En el ámbito profesional comparar productos con otros competidores es esencial para crecer y mejorar constantemente. El término benchmarking se apega a la descripción anterior y se define como un proceso sistemático y continuo para evaluar productos, servicios y procesos con el propósito de mejorar el rendimiento organizacional manteniendo un enfoque estructurado, riguroso, replicable y comparativo [9].

El benchmarking puede cambiar su enfoque teniendo en cuenta el tipo, como formal o informal, y sus clasificaciones, como la naturaleza del referente, contenido y el propósito de la relación [9].

Es una herramienta para gestionar la competitividad de las empresas, debido a que permite integrar mejores prácticas y adoptar nuevas tecnologías. Utilizado para evaluar la competitividad, identificar ventajas y debilidad, e informar estrategias [10].

Para lograr un benchmarking de manera exitosa se requiere seguir una serie de etapas como planificación, análisis, integración, acción y madurez. Además, se necesita disponibilidad de productos, servicios, o procesos comparativos similares [9].

La presente investigación pretende comparar APIs y microservicios idénticos en tres frameworks diferentes, teniendo una funcionalidad similar, los cuales son necesarios para realizar un benchmarking equitativo.

2.2 Lenguajes de programación

A continuación, se presenta una lista de los lenguajes de programación que se destacan en el manejo de alta concurrencia y se diferencian por sus modelos de concurrencia nativos.

2.2.1 Go

Conocido también como Golang, es un lenguaje de programación moderno, compilado, de código abierto con un tipado estricto. Diseñado para tener un código claro, sencillo y fácil de mantener [11][12].

Fue creado por Google a finales de 2007 para abordar los desafíos de los sistemas distribuidos y la necesidad de contar con un lenguaje diseñado para un entorno que cuenta con miles de ingenieros trabajando en una base de código compartida. En 2009, se lanzó

Go como código abierto. En 2012, aparece la versión Go 1, con un compromiso público de compatibilidad retroactiva y bibliotecas estándar [11][13].

Go se destaca por su suporte nativo para la concurrencia, diseñado para manejar los procesadores multinúcleo. Implementa el modelo CSP (Communicating Sequential Processes), goroutines, channels (canales) y un scheduler (planificador) para manejar la concurrencia [11][12][13].

En ingeniería de software a gran escala brinda numerosos beneficios, como, un alto rendimiento y eficiencia, facilidad de aprendizaje, simplicidad, gestión de memoria eficiente, implementación de sistemas escalables, herramientas integradas y ecosistema, seguridad y fiabilidad. Sin embargo, algunos desafíos son la ausencia histórica de características, riesgos de concurrencia, gestión de punteros inseguros y sombra de variables [11][13].

2.2.2 JavaScript

Conocido simplemente como JS, fue creado con el propósito de dar vida a una página web. Surgió como un lenguaje de scripting simple en la década de 1990 para una interactividad mayor en las páginas web [14][15].

El código se escribe en texto plano y no requiere formación especial, es un lenguaje seguro debido a que no utiliza herramientas de manipulación, procesadores o memoria de bajo nivel. Además, evolucionó a Full-Stack, teniendo la capacidad de utilizar el mismo lenguaje tanto para frontend como en backend [14][15].

El utilizar este lenguaje permite la unificación del lenguaje (Full-Stack) con el uso de Node.js que se implementó en 2009, contando con un alto rendimiento y asincronía, ecosistema extenso y herramientas, marcos de trabajos (frameworks) modernos y versatilidad de aplicaciones [14][15].

2.2.3 Elixir

Lenguaje de programación funcional moderno que se ejecuta en la máquina virtual de Erlang (BEAM), conocido por su robustez para desarrollar sistemas de servidor distribuidos a prueba de fallos y altamente disponibles. Diseñado para enfrentar los desafíos de la escalabilidad, concurrencia y tolerancia a fallos [16].

Fue creado por José Valim y concebido como un proyecto de código abierto colaborativo. Cuenta con características de otros lenguajes como Erlang, Clojure y Ruby

para modernizar y mejorar la experiencia de desarrollo en sistemas basados en Erlang [16].

Elixir tiene beneficios clave, en donde se destacan la concurrencia, escalabilidad, tolerancia a fallos, toolset integrado, sintaxis y código simplificado. Sin embargo, algunos desafíos para tener en cuenta son la velocidad, el ecosistema o madurez de librerías y procesos inusuales [16].

2.2.4 Modelos de concurrencia

Como se mencionó en el punto anterior, Go fue diseñado para facilitar la ejecución paralela de tareas y la concurrencia. El funcionamiento del modelo de concurrencia de Go se basa en tres componentes, los cuales según Cox et al. [13] son:

1. Goroutines
    - Hilos de control concurrentes ligeros que se ejecutan en un espacio de direcciones compartido.
    - Comienzan con un tamaño de pocos Kb de pila de memoria y se redimensiona según sea necesario.
    - Al realizar una operación de bloqueo, solo se bloquea esa goroutine, no todo el hilo del sistema subyacente, además permite ejecutar miles o millones de goroutines debido a que son más baratas que los hilos.
    - La abstracción del runtime es la Entrada/Salida (E/S) de bloqueo, sin necesidad de callbacks.
    - Funcionan como procesos autónomos y ligeros que realizan una función específica [12].
2. Channels
    - Es un conducto unidireccional, de tamaño limitado que transporta mensajes tipados que permiten la comunicación y sincronización entre goroutines [12].
    - Tiene un enfoque de compartir memoria al momento de la comunicación.
    - El envío de datos es como una sincronización, donde se pasa la propiedad de los datos del emisor al receptor.

- Cuando no tiene capacidad para almacenar un valor, el envío se bloquea, creando un mecanismo de contrapresión [12].

3. Scheduler y Runtime [12]

   - El runtime tienen su propio gestor de hilos con soporte nativo para la concurrencia.
   - El planificador es un mecanismo de orquestación que abstrae los goroutines de los hilos, permitiendo un trabajo más eficiente.
   - La directiva select se la utiliza para la orquestación y permite controlar la ejecución.

El modelo de concurrencia de JavaScript se caracteriza por su modelo E/S de un solo hilo controlado por eventos asíncronos [17].

1. Naturaleza asíncrona y E/S no bloqueante

- El modelo se basa en el manejo de eventos (event-driven) y en las características de E/S no bloqueante. Cuando se invoca una operación que requiere E/S, la ejecución del programa regresa inmediatamente a la función llamadora para continuar ejecutando la siguiente instrucción. Además, Node.js no cuenta con bibliotecas de programas de bloqueo y aprovecha al máximo la CPU cuando existen un gran número de solicitudes, logrando una eficiencia del proceso [17].

2. Event-Driven

   - Ciclo de Eventos (Event Loop): Cuando se ejecuta un programa, se crea un solo hilo al que se le asigna una cola de eventos, así este hilo ejecuta el ciclo de eventos sin interrupción [17].
   - Dominancia Asíncrona: Las operaciones de E/S se manejan mediante una función de orden superior que acepta una función callback [17].

3. Concurrencia

- El modelo tradicional usa multithreading para resolver la alta concurrencia; sin embargo, Node.js utiliza la solución child_process (proceso hijo) para abordar el problema de la computación pesada en un solo hilo, además aborda el problema de la utilización de múltiples núcleos mediante la comunicación entre procesos y mejora la solidez de la aplicación [17].

Según [16], Elixir se caracteriza por:

- Aislamientos y procesos ligeros
- Concurrencia manejada por el proceso de Erlang, los cuales son extremadamente ligeros y económicos, con un costo inicial de 1 a 2 Kb que se crean en microsegundos. Están aislados y no comparten memoria, lo que conlleva a que solo afecte a un proceso en caso de una caída.
- Comunicación por Mensajes
    - Los procesos se comunican mediante mensajes asíncronos, debido a que no comparten memoria.
    - Un mensaje es cualquier término de Elixir y es copiado profundamente del remitente al buzón del receptor.
    - Cada proceso tiene un buzón que funciona como una cola FIFO (primero en entrar, primero en salir).
    - La recepción de un mensaje se realiza mediante la construcción receive, la cual extrae un mensaje del buzón, y lo empareja con patrones definidos, y ejecutar el código correspondiente.
    - Cuando un proceso envía un mensaje de forma sincrónica, debe esperar una respuesta. La implementación de la comunicación sincrónica (solicitud-respuesta) se construye manualmente sobre el mecanismo asíncrono básico, incluyendo el pid (identificador del proceso) del remitente en el mensaje inicial y esperando la respuesta.
- Planificador y Escalabilidad
    - La BEAM utiliza sus propios planificadores para distribuir la ejecución de los procesos sobre los núcleos de CPU disponibles, maximizando el paralelismo.
    - El planificador es preventivo, debido a que asigna una pequeña ventana de ejecución a cada proceso y luego lo pausa para ejecutar el siguiente. Las conmutaciones de contexto son frecuentes, ocurriendo en menos de 1 milisegundo. Esta preventividad garantiza que una tarea larga que consuma CPU no bloquee al resto del sistema.

- Las operaciones de E/S se delegan internamente a hilos separados (async threads) o a servicios del kernel, lo que significa que un proceso que espera E/S no bloquea la ejecución de otros procesos.
- La recolección de basura (garbage collection) se realiza a nivel de proceso individual, reduciendo las pausas en todo el sistema y mejorando la capacidad de respuesta.

- Tolerancia a fallos y supervisores
  - La tolerancia a fallos se logra mediante el aislamiento de procesos y el uso de enlaces y monitores para detectar cuando un proceso termina.
  - Los supervisores son procesos que supervisan a otros procesos (workers) y los reinician si fallan, lo que permite que el sistema se recupere automáticamente de errores inesperados.

2.3 API y Microservicio

2.3.1 API

Una API es un componente esencial en la transformación y el entorno digital actual, ya que permite la interoperabilidad entre distintos sistemas o capas de una misma aplicación. Facilitan la comunicación entre diversas aplicaciones sin importar la plataforma o el lenguaje de programación utilizado [4].

Permiten interoperabilidad y adaptación, exposición de servicios y facilitación de negocios [4][1]. Existen algunos tipos de APIs:

- API RESTful que tienen la capacidad de conectar aplicaciones y dispositivos de manera eficiente y promueve la independencia cliente-servidor, mejorando la escalabilidad [4].
- GraphQL es una alternativa para mejorar problemas con el acceso a datos y la gestión de versiones en APIs basadas en REST. Ofrece mayor flexibilidad al obtener datos, optimizando eficiencia y evitando la creación de múltiples endpoints [4].
- gRPC (Google Remote Procedure Call) permite la comunicación entre aplicaciones y servicios de manera independiente del lenguaje y la plataforma. Es

la mejor opción en rendimiento debido al manejo de la baja latencia y alta escalabilidad [18].

2.3.2 Microservicio

La Arquitectura de Microservicios (MSA) es un conjunto de servicios pequeños, independientemente desplegables, autónomos, autocontenidos y débilmente acoplados que se comunican entre si a través de APIs de recursos HTTP [1][3].

Los microservicios brindan escalabilidad y flexibilidad, tolerancia a fallos, desarrollo y mantenimiento independiente, y heterogeneidad tecnológica. Sin embargo, existen algunos desafíos como la gestión, debido a que adoptar microservicios introduce complejidad a la gestión y seguridad [4][1][3].

Teniendo en cuenta lo anterior, la elección entre una arquitectura monolítica o MSA depende de las necesidades y objetivos específicos de cada proyecto.

2.4 Atributos de Calidad del Sistema

Los sistemas de software, con la evolución de la tecnología, exigen criterios de calidad más estrictos para garantizar su competitividad y sostenibilidad en el mercado. Por esa razón, es necesario definir algunos atributos no funcionales para entender con mayor profundidad la presente investigación.

2.4.1 Rendimiento

Es la eficiencia en el procesamiento de datos y la computación del sistema dando como resultado menor tiempo de cálculo o procesamiento y menor dependencia de recursos de hardware. En escenarios de alta concurrencia y tiempo real, el rendimiento recae en los algoritmos, afectando directamente la experiencia de usuario, seguridad del sistema y la continuidad del servicio [19].

2.4.2 Escalabilidad

Es la capacidad de un sistema para mantener o mejorar su rendimiento mediante la adición de recursos ante un aumento de la carga de trabajo [20]. Existen dos tipos de escalabilidad:

- Escalamiento Horizontal consiste en añadir más nodos de cómputo o servidores, considerado el más flexible y escalable [20].

- Escalamiento Vertical consiste en aumentar los recursos de un nodo individual como la CPU, memoria o disco. Sin embargo, está limitada por la configuración máxima del hardware [20].

Se ve afectada por varios factores, entre ellos tenemos, el diseño de la arquitectura, selección de protocolos, el equilibrio de carga, la consistencia de datos, restricciones de ancho de banda y latencia de red a medida que el sistema crece [20].

2.4.3 Disponibilidad

Es la capacidad de un sistema para permanecer operativo y accesible, incluso bajo condiciones de estrés o falla. Es esencial para proporcionar una experiencia de usuario ininterrumpida [21].

Existen algunas estrategias para mejorar la disponibilidad, entre las cuales tenemos, la redundancia, mecanismos de conmutación por error y autoescalamiento [21].

2.4.4 Concurrencia

Describe como se organizan y ejecutan múltiples tareas. Tiene dos tipos de modelos, la concurrencia entrelazada y la verdadera concurrencia. Su naturaleza implica que las acciones de dos ramas paralelas pueden tener causalidades o conflictos entre sí [22].

2.5 Infraestructura en la nube

Es una disciplina centrada en el diseño, la administración y el mantenimiento de servicios y sistemas tecnológicos en la nube. La nube o computación en la nube se basa en recursos informáticos que se pueden acceder a través de internet y son configurables y pueden ser aprovisionados rápidamente, requiriendo un mínimo esfuerzo de gestión por el proveedor del servicio. Se define como un modelo que permite el acceso bajo demanda [23].

Ofrece varias ventajas, en donde destacan la escalabilidad, flexibilidad, reducción de costos operativos, seguridad, eficiencia operativa, optimización de recursos, entre otros. Sin embargo, también existen limitaciones o desafíos, como la seguridad y privacidad, integración con sistemas heredados, falta de personal capacitado, costos de implementación, resistencias al cambio, entre otros [23][24].

En los últimos años ha influido positivamente en diversos sectores y está siendo fundamental para la modernización empresarial y la transformación digital, algunos usos

destacados están en la gestión de servicios de TI, el sector educativo y financiero, en si en aquellas instituciones que buscan optimizar recursos y mejorar procesos [23][24].

El entorno de infraestructura en la nube fue elegido para el despliegue de los servicios y para el desarrollo del benchmarking debido a su capacidad de configuración y así mantener condiciones homogéneas durante la evaluación de las APIs y microservicios.

2.6 Trabajos relacionados

A continuación, se presentan un compendio de los trabajos relacionados, en donde encontramos distintos enfoques en la comparación de frameworks, desde comparar un aspecto especifico hasta realizar una comparación de varios aspectos, que permiten tener una perspectiva mucho más amplia y completa del benchmarking.

Gallego [25] realizó una comparación entre PHP (Laravel), Python (Flask), Java (Spring Boot) y JavaScript (Express), evaluando el rendimiento, la facilidad de despliegue y la agilidad en el ciclo de desarrollo. La metodología se fundamentó en la creación de aplicaciones funcionales idénticas para la gestión de contenido multimedia, las cuales fueron encapsuladas en contenedores Docker y sometidas a pruebas de latencia en operaciones de registro y consulta de datos bajo una infraestructura de hardware estandarizada. JavaScript con Express se consolidó como el motor de ejecución más ágil para APIs REST, mientras que PHP presentó la mayor latencia en servicios web; no obstante, PHP compensó su menor rendimiento con una eficiencia de desarrollo superior gracias a la facilidad sintáctica de Laravel.

Yánez y Jaramillo [26] realizaron una revisión bibliográfica sistemática y comparativa entre los ecosistemas Spring y Struts. Se desglosó las fortalezas, debilidades y contribuciones técnicas de ambos marcos de trabajo. Se empleo una metodología de búsqueda rigurosa en bases de datos académicas, aplicando criterios de inclusión y exclusión; el análisis abordó dimensiones críticas como la arquitectura modular, el rendimiento operativo bajo carga, la curva de aprendizaje y la vigencia en el mercado actual. Spring fue superior en tiempos de respuesta, escalabilidad horizontal y eficiencia en el uso de recursos ante solicitudes concurrentes, mientras que, Struts mantiene su relevancia en sistemas legados debido a su estructura MVC, menor flexibilidad en entornos modernos y una curva de aprendizaje más pronunciada. La limitación principal

fue la selección de fuentes que podrían no haber abarcado la totalidad de la literatura existente.

Lara [27] realizó una comparativa técnica entre los stacks Java (Spring Boot) e JavaScript (MERN) para determinar la mejor tecnología en la implementación de servicios web RESTful destinados a la consulta de datos en el entorno de la PUCESE. Se desarrollaron dos microservicios funcionalmente equivalentes bajo una arquitectura orientada a recursos (ROA); se desplegó en contenedores Docker y se ejecutaron pruebas de carga, estrés y concurrencia empleando Apache JMeter y Siege. Se evaluaron la usabilidad y eficiencia de la herramienta. JavaScript ofrece una usabilidad ligeramente superior y una mayor agilidad en tiempos de respuesta durante pruebas de carga moderada, el stack de Java demostró una robustez y estabilidad mayores al ser sometido a pruebas de estrés de alta concurrencia. Como limitaciones, el estudio trató con un volumen de datos inicial de 5,000 registros y a un entorno de hardware local específico.

Hussein et al. [28] realizaron una evaluación comparativa de los lenguajes de programación, Go, Java, Python y Node.js, en el contexto de la arquitectura de microservicios. Se empleó una metodología de análisis sistemático basada en la recopilación de datos empíricos, métricas de referencia de la industria y una revisión de literatura técnica, evaluando la velocidad de ejecución, el manejo de concurrencia, el soporte de ecosistemas y la facilidad de integración mediante contenedores. Go destacó por su baja latencia y alta concurrencia. Java demostró robustez para sistemas empresariales, Python sobresalió en rapidez de desarrollo y tareas de ciencia de datos, y Node.js en servicios intensivos en E/S en tiempo real. Las limitaciones fueron la profundidad de los benchmarks de seguridad específicos y la falta de análisis sobre lenguajes emergentes.

Di Meglio y Starace [29] compararon cinco frameworks REST (Spring Boot, Micronaut, Express, Nest y Django) sobre diversos entornos de ejecución, para determinar su impacto en el rendimiento, consumo de recursos y eficiencia energética. Se utilizó un prototipo industrial de API REST implementado equivalentemente en 12 configuraciones distintas, las cuales fueron sometidas a pruebas automatizadas de carga y estrés en entornos controlados mediante Docker y Apache JMeter, monitoreando la latencia, el rendimiento, el uso de CPU/RAM y las emisiones de CO2. Las configuraciones basadas en JavaScript/TypeScript, alcanzaron el rendimiento superior en términos de velocidad de respuesta y capacidad de procesamiento. Los entornos Python

mostraron una mayor eficiencia relativa en el consumo de recursos básicos. Concluyendo que la elección del entorno de ejecución puede alterar drásticamente el perfil de eficiencia energética del framework. Las limitaciones fueron el uso de cargas sintéticas y una única aplicación de prueba.

Grudniak y Dzieńkowski [30] compararon dos ecosistemas de JavaScript: Express y Hapi, para la creación de aplicaciones de servidor. Se diseñó dos servicios backend idénticos y se usó un cliente en React y una base NoSQL (MongoDB). Se evaluó el desempeño de las operaciones CRUD bajo cuatro escenarios de carga de datos distintos, desde cadenas de texto hasta colecciones de objetos complejos. Express tuvo un rendimiento superior a Hapi, registrando sistemáticamente menores tiempos de latencia en la mayoría de las operaciones, a excepción del cuarto escenario en donde el rendimiento fue idéntico. Las limitaciones fueron centrar las pruebas únicamente en rendimiento y dejar de lado otros aspectos.

Ardiansyah y Fatwanto [31] compararon el consumo de memoria RAM entre JavaScript (Node.js) y Golang en el desarrollo de APIs REST, para identificar cuál ofrece mayor eficiencia de recursos para dispositivos con capacidades limitadas. Emplearon una metodología cuantitativa experimental basada en la creación de servicios backend equivalentes que ejecutaban operaciones de recuperación de datos utilizando dos conjuntos de datos distintos de 100 000 registros cada uno. El análisis estadístico se llevó a cabo mediante pruebas de normalidad, la prueba de Wilcoxon y pruebas t para datos emparejados, evaluando el desempeño en entornos locales y remotos. JavaScript registró promedios ligeramente inferiores a Golang en la mayoría de los escenarios, concluyendo que ambas tecnologías son altamente competitivas y eficientes para el desarrollo de servicios móviles. Las limitaciones fueron el uso único del método GET y la métrica de memoria, omitiendo el análisis de otras operaciones CRUD y el uso de CPU.

Lion et al. [32] compararon el rendimiento entre lenguajes gestionados (Java, Go, JavaScript y Python) y un lenguaje de sistemas (C++). Implementaron una metodología de benchmarking mediante la instrumentación de runtimes y análisis cuantitativo a nivel de bytecode, utilizando la suite LangBench para aislar y medir el costo computacional de características de los lenguajes, como la recolección de basura, la comprobación de límites de arreglos y la sobrecarga de los intérpretes frente a los compiladores JIT. JavaScript y Python son consistentemente más lentos debido a la naturaleza dinámica y al peso de la interpretación de sus runtimes, Java y Go pueden igualar o incluso superar

a C++ en escenarios específicos gracias a optimizaciones en tiempo de ejecución. La investigación se limitó al análisis del tiempo de ejecución y la sobrecarga del runtime.

Brar et al. [33] evaluaron el rendimiento de tecnologías utilizadas en el desarrollo backend: Node.js, PHP y Python (Django), enfocándose en su capacidad para manejar aplicaciones de alta concurrencia y gran volumen de datos. Combinó pruebas de benchmark sintéticos y pruebas de escenario, basadas en el comportamiento del usuario, para medir métricas críticas como el tiempo de respuesta, el rendimiento y la estabilidad bajo diferentes niveles de carga de trabajo. Node.js se destaca en eficiencia para sitios web con alta intensidad de E/S. PHP se identificó como una solución para aplicaciones de escala pequeña a mediana, mientras que Python destacó por su facilidad de desarrollo en estructuras web complejas, aunque los dos con un rendimiento inferior en concurrencia frente a Node.js. La limitación del estudio fue el uso de métricas de ejecución externas sin profundizar en la evaluación de recursos internos.

Durante la preparación de este trabajo, el autor utilizó Consensus para la búsqueda de literatura académica, NotebookLM para resumir los artículos seleccionados, y Gemini para mejorar la redacción y extraer información relevante de los artículos. Tras el uso de estas herramientas, el autor revisó, validó y editó el contenido para asegurar la precisión de los datos frente a las fuentes originales, y asume la responsabilidad total por el contenido final de esta publicación.

CAPÍTULO III: MATERIALES Y MÉTODOS

3.1 Enfoque y tipo de investigación

La presente investigación tiene un enfoque cuantitativo. Se fundamenta en la medición numérica y el análisis estadístico para describir y comparar el comportamiento de distintos objetos de estudio, como lo son los frameworks de desarrollo, bajo condiciones controladas y repetibles. Los datos recolectados son métricas de rendimiento objetivas y cuantificables. La recolección de datos se la realizó con instrumentos de medición automatizados, y el análisis se apoya en estadística descriptiva. Este tipo de enfoque nos permite establecer una comparación objetiva y reproducible.

La investigación es de tipo experimental y comparativa, con elementos descriptivos:

- Experimental debido a la manipulación de variables independientes para observar y medir el efecto que produce sobre las variables dependientes. El resto de las condiciones se mantienen constantes, como es el caso de la lógica de negocio, el entorno, la herramienta de carga, pruebas, entre otros; con el fin de aislar la variable de interés y observar las diferencias en los resultados.
- Comparativa por contrastar el rendimiento obtenido por los frameworks, evaluando cada uno bajo las mismas condiciones experimentales. Buscando establecer las diferencias entre las alternativas seleccionadas.
- Descriptivo por documentar y describir el comportamiento de cada framework en cada escenario de prueba, antes de pasar a realizar las comparaciones y análisis correspondientes.

Se trata de una investigación experimental-comparativa de tipo transversal que combina la descripción de métricas y la comparación sistemática basada en el diseño experimental.

3.2 Diseño de la investigación: Materiales, equipos y software

El estudio adopta un diseño experimental comparativo, donde la variable independiente es el framework de desarrollo, utilizado para implementar las APIs y microservicios. Los frameworks evaluados son Express, Gin y Phoenix. La comparación se realiza en dos contextos de arquitectura:

- **Arquitectura monolítica**: Cada framework implementa dos APIs REST: CRUD de Usuarios y CRUD de Facturas, con la misma lógica de negocio y una base de datos PostgreSQL.
- **Arquitectura de microservicios**: Cada framework implementa únicamente un microservicio: CRUD de Usuarios, en el que todas las peticiones externas se enrutan a través de un API Gateway desarrollado con FastAPI (Python).

Las variables dependientes corresponden a las métricas de rendimiento seleccionadas de la norma ISO/IEC 25023. Las variables controladas que se mantienen constantes para todos los frameworks incluyen la lógica de negocio, el esquema de la base de datos, el entorno de despliegue, el método de despliegue y las pruebas aplicadas mediante k6. Estas variables se describen en el apartado 3.3.

Como se mencionó en la sección anterior, este estudio es de tipo transversal, debido a que la recolección de datos se efectuó en una única ventana de tiempo. Para mitigar la variabilidad del entorno de producción simulado en la nube y garantizar la confiabilidad estadística, se ejecutaron tres réplicas para cada escenario y tipo de prueba. El análisis considera el promedio de las tres ejecuciones, mediante estadística descriptiva y tablas comparativas para evaluar el comportamiento de los frameworks.

A continuación, en la **Tabla 1**, se detallan los componentes empleados en el desarrollo de la investigación.

Tabla 1 *Materiales, equipos y software empleados en la investigación.*

Materiales	**Equipos**	**Software**
Scripts de prueba de carga y estrés	Laptop personal	Frameworks: Express, Gin, Phoenix
Código fuente de las APIs	Servidores de Render (Plan Starter)	Contenerización: Docker
Código fuente del API Gateway		Base de datos: PostgreSQL
Scripts de despliegue		Herramienta de pruebas: k6
Archivos JSON, TXT y HTML generados por k6		Control de versiones: Git / GitHub

3.3 Métodos, técnicas e instrumentos.

En esta sección se describen las unidades experimentales, el conjunto de pruebas ejecutadas, los instrumentos de medición y la operacionalización de las variables.

3.3.1 Unidades experimentales y tamaño del experimento

La investigación evalúa un total de nueve unidades experimentales o configuraciones tecnológicas, distribuidas en dos contextos de arquitectura:

- 3 frameworks (Express, Gin, Phoenix) × 2 APIs (Usuario y Factura) = 6 configuraciones.
- 3 frameworks (Express, Gin, Phoenix) × 1 microservicio (Usuario) = 3 configuraciones.

Para cada configuración se definieron dos pruebas: una prueba de carga y una prueba de estrés. Con el propósito de mitigar la variabilidad del entorno en la nube y mejorar la confiabilidad de los resultados, cada prueba se ejecutó tres veces. Por lo tanto, el tamaño total del experimento es de 54 pruebas independientes (9 configuraciones × 2 pruebas × 3 réplicas).

Se trata de un experimento controlado en el que cada configuración se evalúa directamente bajo las mismas condiciones. Los resultados obtenidos en estas 54 ejecuciones constituyen la totalidad de los datos primarios. El análisis de los datos se realizará mediante estadística descriptiva, utilizando el promedio de las tres réplicas y tablas comparativas para evaluar el comportamiento de los frameworks.

3.3.2 Técnicas e instrumentos de recolección de datos

Se emplearon las siguientes técnicas e instrumentos estructurados para garantizar un proceso automatizado, controlado y reproducible:

- **Pruebas de carga y estrés automatizadas con k6**: La herramienta de código abierto k6 se utilizó para simular y generar tráfico HTTP hacia las APIs desplegadas en Render. Cada una de las pruebas ejecutadas generó de forma automatizada un archivo de salida en formato JSON con el desglose de las métricas de rendimiento. Además, dos archivos complementarios: uno HTML y

otro TXT, el primero muestra un resumen con gráficos y tablas; y el segundo guarda toda la información mostrada en la consola durante la prueba.

- **Monitoreo de recursos en Render**: Durante el tiempo de ejecución de cada prueba se accedió al panel de monitoreo del servicio en Render, el cual muestra el uso porcentual de CPU y la memoria RAM consumida en tiempo real. Estos valores fueron registrados mediante capturas de pantalla y observación directa en los momentos de mayor demanda.
- **Registro de datos primarios**: Los archivos JSON generados por k6 y los valores máximos del consumo de recursos obtenidos del panel de Render conforman los datos primarios, los cuales fueron analizados y promediados, y registrados en una tabla para una mayor comprensión.

3.3.3 Matriz de operacionalización de variables.

A continuación, se presenta la operacionalización de las variables que intervienen en la investigación.

Variable independiente

Se trata de la manipulación del ecosistema de desarrollo a través de tres opciones tecnológicas con diferentes arquitecturas de ejecución y lenguajes de programación, esto se puede observar en la **Tabla 2**.

Tabla 2 *Operacionalización de variable independiente*

Denominación	Definición	Categorías
Framework de desarrollo	Entorno de software y lenguaje de programación utilizado para codificar la lógica de negocio de las APIs y microservicios.	Express Gin Phoenix

Variables dependientes

Corresponden a las métricas de rendimiento de la ISO/IEC 25023:2016, que cambian en función del framework evaluado, en este caso se seleccionó el apartado 8.3 denominado Medidas de eficiencia del rendimiento (*Performance efficiency measures*)

que cuenta con 12 métricas, y también se definió la herramienta o prueba con la cual se va a obtener los datos. Las métricas seleccionadas son mostradas en la **Tabla 3**.

Tabla 3 *Operacionalización de variables dependientes[34]*

ID	**Métrica**	**Función de Medición**	**Herramienta**
PTb-1-G	Tiempo medio de respuesta (*Mean response time*)	$X = \sum_{i=1}^{n} \frac{(A_i)}{n}$ A_i = Tiempo empleado por el sistema para responder a una tarea especifica del usuario o del sistema en la medición *i-ésima*. n = Número de respuestas medidas.	k6 (Prueba de Carga)
PTb-2-G	Adecuación del tiempo de respuesta (*Response time adequacy*)	$X = \boldsymbol{A/B}$ A = Tiempo medio de respuesta medido por PTb-1-G. B = Tiempo de respuesta objetivo especificado.	k6 (Prueba de Carga)
PTb-5-G	Rendimiento medio (*Mean throughput*)	$X = \sum_{i=1}^{n} \frac{A_i/B_i}{n}$ A_i = Número de trabajos completados durante el *i*-ésimo período de observación. B_i = Duración del *i*-ésimo período de observación. n = Número de observaciones.	k6 (Prueba de Carga)
PRu-1-G	Utilización media del procesador (*Mean processor utilization*)	$X = \sum_{i=1}^{n} \frac{A_i/B_i}{n}$ A_i = Tiempo de procesador realmente utilizado para ejecutar un conjunto determinado de tareas en la observación *i*. B_i = Tiempo de operación necesario para realizar las tareas en la observación *i*. n = Número de observaciones.	Panel de Render
PRu-2-G	Utilización media de memoria	$X = \sum_{i=1}^{n} \frac{A_i/B_i}{n}$	Panel de Render

	(*Mean memory utilization*)	A_i = Cantidad de memoria realmente utilizada para ejecutar un conjunto determinado de tareas durante el procesamiento de la muestra *i*. B_i = Cantidad de memoria disponible para realizar las tareas durante el procesamiento de la muestra *i*. n = Número de muestras procesadas.	
PRu-4-S	Utilización del ancho de banda (*Bandwidth utilization*)	X = $\boldsymbol{A/B}$ A = Ancho de banda real de transmisión medido durante el tiempo necesario para ejecutar un conjunto determinado de tareas. B = Capacidad de ancho de banda disponible para ejecutar un conjunto determinado de tareas.	k6 (Prueba de Carga)
PCa-1-G	Capacidad de procesamiento de transacciones (*Transaction processing capacity*)	X = $\boldsymbol{A/B}$ A = Número de transacciones completadas durante el período de observación. B = Duración del período de observación.	k6 (Prueba de Estrés)
PCa-2-G	Capacidad de acceso de usuarios (*User access capacity*)	X = $\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \frac{\boldsymbol{A_i}}{\boldsymbol{n}}$ A_i = Número máximo de usuarios que pueden acceder simultáneamente al sistema en la observación *i*. n = Número de observaciones.	k6 (Prueba de Estrés)
PCa-3-S	Adecuación al incremento del acceso de usuarios (*User access increase adequacy*)	X = $\boldsymbol{A/B}$ A = Número de usuarios añadidos correctamente durante el período de observación. B = Duración del período de observación.	k6 (Prueba de Estrés)

Para obtener la métrica de adecuación del tiempo de respuesta, es necesario mencionar que se utilizó una alternativa, la cual se describe en el aparado Medidas de eficiencia del rendimiento, que consiste en usar percentiles en vez del tiempo medio de respuesta. Por lo tanto, en la investigación se empleó el percentil 95 (P95) como medida de referencia, debido a que permite representar de manera más precisa la latencia durante las pruebas de carga.

Variables controladas

Son los factores que se mantienen constantes para asegurar que las diferencias en el rendimiento se deban exclusivamente al framework evaluado.

Tabla 4 *Operacionalización de variables controladas*

Variable controlada	Valor constante
Lógica de negocio y endpoints	CRUD de Usuarios y Facturas (solo en monolíticos) idénticos en todos los frameworks
Base de datos	PostgreSQL 18 alojada en Render (256 MB RAM, 0.1 CPU), con la misma instancia y esquema
Infraestructura de servidor	Render plan Starter (512 MB RAM, 0.5 CPU)
Método de despliegue	Despliegue con repositorio GitHub e imagen construida a partir de Dockerfile
Herramienta de carga	k6 con la misma configuración de pruebas de carga y estrés dependiendo de la arquitectura
Máquina cliente	Laptop (Dell Inc. / Inspiron 5420) con conexión a internet estándar
Condiciones de red	Pruebas ejecutadas en el mismo día, en horarios cercanos, para minimizar la variabilidad de internet

3.4 Procedimiento y análisis de datos.

3.4.1 Procedimiento experimental

El desarrollo del experimento se estructuró en etapas secuenciales, ejecutadas de forma sistemática para cada uno de los tres frameworks: Express, Gin y Phoenix; dentro de los dos contextos de arquitectura establecidos:

- **Preparación del código fuente y despliegue:** Se implementó la lógica de negocio homóloga en cada framework, CRUD de Usuarios y Facturas para los monolitos

y CRUD de Usuarios para los microservicios, asegurando que los endpoints, modelos de datos y conexiones a PostgreSQL fueran funcionalmente equivalentes. El código fuente se alojó en repositorios Git independientes por framework y arquitectura. Cada repositorio incluía un Dockerfile con las instrucciones para construir la imagen del servicio. A través de la plataforma Render (plan Starter) se vinculó cada repositorio, configurando el despliegue como Web Service y seleccionado la misma región "Oregon (US West)". La base de datos PostgreSQL se aprovisionó como servicio gestionado en Render, con una instancia compartida para todos los frameworks. La base de datos utilizada en Render fue la predeterminada por el plan gratuito.

- **Ejecución de las pruebas de carga y estrés:** Con el servicio desplegado y verificado, mediante una petición de prueba para asegurar disponibilidad, se procedió a ejecutar las pruebas piloto simples desde el equipo personal utilizando k6, una vez comprobada su correcta funcionalidad, se definieron dos tipos de prueba:

- **Prueba de carga:** Simula un flujo constante de usuarios virtuales (VUs) durante un período fijo, con una rampa de subida gradual, con el objetivo de medir el comportamiento bajo una demanda sostenida. Los valores usados en estas pruebas se observan en la **Tabla 5**.

- **Prueba de estrés:** Incrementa progresivamente los VUs hasta un pico máximo, utilizando rampas escalonadas, para identificar el punto de degradación del servicio. Los valores usados en estas pruebas se observan en la **Tabla 5**.

Tabla 5 *Configuración de Pruebas de Carga y Estrés*

Arquitectura	Prueba de Carga	Prueba de Estrés
Monolítico	200 VUs constantes Tiempo: 4 minutos	200 VUs cada 60 segundos hasta 3000 VUs Duración: Degradación del sistema
Microservicio	25 VUs constantes Tiempo: 4 minutos	50 VUs cada 20 segundos hasta 750 VUs Duración: Degradación del sistema

Un ejemplar de los scripts de las pruebas de carga y estrés utilizados, se encuentran en la sección de ANEXOS, en donde se puede ver a detalle toda la codificación y las configuraciones realizadas para cada arquitectura.

Para asegurar la validez de los resultados frente a la volatilidad de la infraestructura en la nube, cada prueba se replicó tres veces consecutivas bajo las mismas condiciones controladas. Al finalizar cada réplica, k6 exportó automáticamente un reporte estructurado en formato JSON con las métricas detalladas del comportamiento del tráfico. En paralelo, durante el periodo activo de cada prueba, se registraron las métricas de consumo de hardware, valores máximos de CPU y memoria RAM, directamente desde el panel de monitoreo de Render.

- **Recolección y consolidación de datos:** Los archivos JSON generados por k6 se almacenaron con nombres descriptivos para evitar confusiones. De cada archivo se extrajeron las métricas descritas anteriormente en la **Tabla 3**. Los valores máximos de CPU (%) y memoria RAM (MB) se obtuvieron del panel de Render. Todos estos datos se promediaron y se colocaron en una tabla.

3.4.2 Análisis de datos

El análisis de los datos recolectados es descriptivo y comparativo, utilizando tablas y gráficos para contrastar el comportamiento de los frameworks evaluados. El análisis abarca los siguientes puntos:

- **Estadística descriptiva:** Se obtuvo un promedio de los valores reportados por k6 y Render. Estos valores sirvieron para estabilizar la variabilidad del entorno y construir el conjunto de datos de análisis.
- **Comparaciones directas:** Se elaboraron tablas comparativas donde se observan los valores de un mismo tipo de prueba entre los tres frameworks, segmentando por arquitectura y esta a su vez en las métricas seleccionadas.
- **Visualización gráfica:** Se generaron gráficos para cada métrica, facilitando la identificación visual de la capacidad de cada framework.
- **Interpretación de resultados:** A partir de los resultados observados, se analizan las diferencias y se establecen relaciones con la literatura científica revisada en el desarrollo del capítulo II.

3.4.3 Aspectos éticos

Todas las herramientas de software empleadas (k6, Docker, frameworks, PostgreSQL) son de código abierto o de uso gratuito en sus planes básicos, respetando sus respectivas licencias. Las pruebas automatizadas se realizaron sobre infraestructura propia del investigador, como la cuenta personal de Render en plan Starter y equipo local, sin afectar servicios de terceros ni generar tráfico malicioso. Los datos procesados durante las interacciones CRUD corresponden a registros sintéticos generados algorítmicamente durante la ejecución de cada prueba, garantizando la ausencia de datos reales o sensibles. De esta forma, la investigación se rige estrictamente bajo los principios de honestidad científica, objetividad en la manipulación de las pruebas, transparencia metodológica y reproducibilidad del experimento.

CAPÍTULO IV: RESULTADOS Y ANÁLISIS

Este capítulo presenta los resultados durante la fase de desarrollo de la investigación, detallando y comparando los resultados de los frameworks por métrica y por arquitectura.

4.1 Arquitectura Monolítica

A continuación, se presentan los resultados de las pruebas realizadas en cada framework de desarrollo, seccionado por las métricas descritas en el anterior capítulo.

4.1.1 Tiempo medio de respuesta

Esta métrica determina la capacidad de un sistema para procesar solicitudes dentro de umbrales tolerables de latencia. Para esta evaluación, se contrastaron el tiempo promedio de respuesta y el P95.

Como se observa en la **Tabla 6**, los resultados globales del tiempo promedio demuestran una clara ventaja operativa por parte de Gin y Phoenix, registrando una media de 196.07 ms y 201.10 ms respectivamente. Esta mínima diferencia en milisegundos evidencia que ambas tecnologías gestionan eficientemente la entrada y salida de datos. En contraste, Express mostró el desempeño menos favorable en todas las operaciones del CRUD, registrando una media de 233.88 ms.

Los datos del P95 ratifican y profundizan el comportamiento de los tres frameworks. Phoenix, en este escenario, demuestra superioridad, seguido de cerca por Gin con una diferencia mínima de milisegundos en la mayoría de las operaciones. Por

otro lado, Express muestra una inferioridad notable, llegando a superar los 400 ms en la mayoría de las operaciones.

Tabla 6 *Tiempo medio de respuesta - Usuario*

ID	Solicitud	Express	Gin	Phoenix	Unidad / Nota
PTb-1-G	POST_avg	234.46	201.18	202.55	ms
	GET_avg	226.25	189.23	197.61	ms
	PUT_avg	240.27	197.48	202.11	ms
	DELETE_avg	234.53	196.38	202.12	ms
	TOTAL_avg	233.88	196.07	201.10	ms
	POST_P95	422.57	276.97	268.34	ms
	GET_P95	378.75	238.45	246.14	ms
	PUT_P95	440.69	289.25	270.84	ms
	DELETE_P95	412.79	286.37	276.48	ms

En la **Figura 2**, se puede visualizar estos datos de una manera más entendible y grafica. En donde se observa el promedio general y los P95.

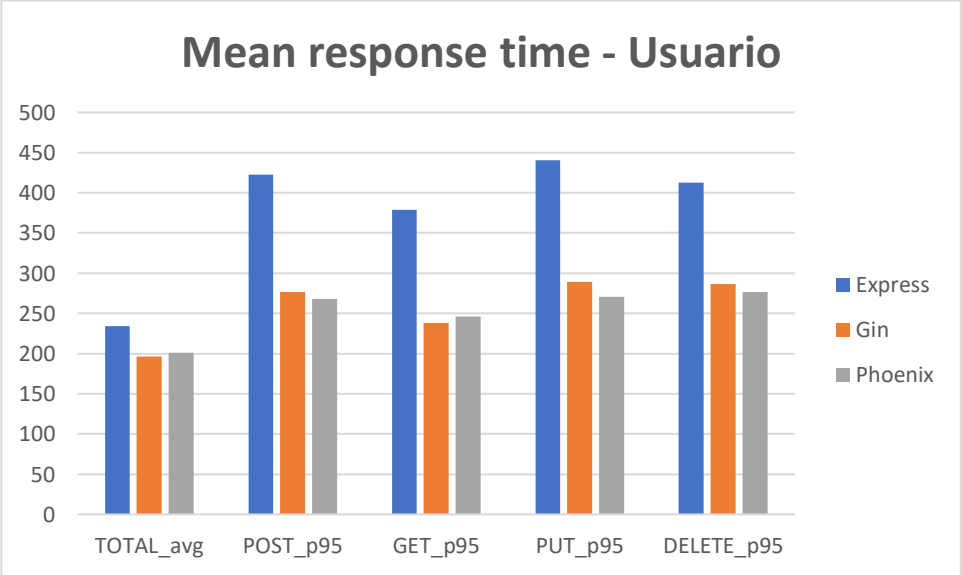


Figura 2 Tiempo medio de respuesta - Usuario

Para evaluar la capacidad de respuesta ante operaciones de mayor complejidad transaccional, se ejecutaron pruebas sobre el CRUD de Facturas.

Al comparar estos resultados, en la **Tabla 7,** con los resultados previos del CRUD de Usuarios, se evidencia una degradación generalizada de los tiempos de respuesta en

los tres entornos evaluados. El framework Phoenix demostró una ligera superioridad ante esta carga, con un tiempo promedio total de 577.15 ms. Gin, con estadísticas similares, obtuvo una media general de 595.25 ms. Por su parte, Express sufrió una penalización considerable en su eficiencia, elevando su media general a 965.05 ms.

La degradación se evidencia aún más en el P95, en donde Phoenix supera los 1000 ms en todas las operaciones. Gin y Express tienen datos muy similares en este apartado, aunque Gin mantiene una leve ventaja con valores que oscilan entre los 1 214 ms y 1 469 ms, frente a los rangos de 1 260 ms y 1 556 ms registrados por Express. Este comportamiento evidencia la vulnerabilidad de los frameworks ante operaciones transaccionales complejas.

Tabla 7 *Tiempo medio de respuesta - Factura*

ID	Solicitud	Express	Gin	Phoenix	Unidad / Nota
PTb-1-G	POST_avg	1007.66	670.28	547.01	ms
	GET_avg	852.07	512.39	548.83	ms
	PUT_avg	1057.97	649.50	708.80	ms
	DELETE_avg	942.49	548.85	510.67	ms
	TOTAL_avg	965.05	595.25	577.15	ms
	POST_P95	1517.75	1469.34	1072.93	ms
	GET_P95	1260.97	1240.06	1058.02	ms
	PUT_P95	1556.96	1368.89	1297.26	ms
	DELETE_P95	1449.94	1214.30	1042.91	ms

En la Figura 3, se presenta de manera gráfica el promedio general y los P95 de los datos anteriores.

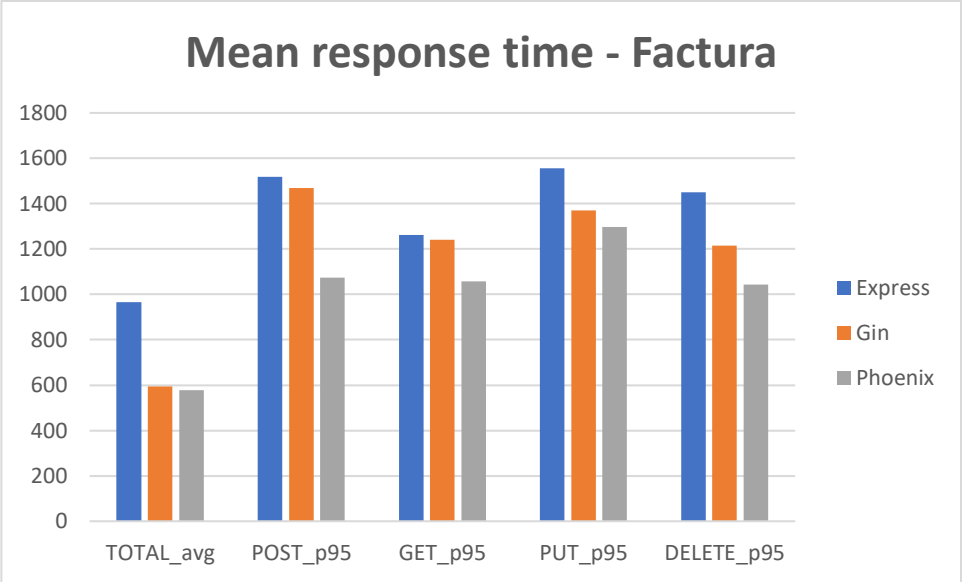


Figura 3 Tiempo medio de respuesta - Factura

4.1.2 Adecuación del tiempo de respuesta

Esta métrica de comportamiento temporal compara el tiempo real medido contra un objetivo establecido. Si el resultado es menor o igual 1, el rendimiento se considera bueno.

Como se observa en la **Tabla 8**, bajo el escenario simple del CRUD de Usuarios, los tres frameworks evaluados cumplieron con éxito la totalidad de los objetivos fijados ($X \leq 1$), los cuales están presentes en la tabla. Los frameworks Phoenix y Gin demostraron los índices más eficientes, manteniendo una diferencia mínima entre sí, en todas las operaciones. Por su parte, Express registró los coeficientes de adecuación más altos, alcanzando su valor más crítico de 0.473 en la solicitud GET, aunque permaneciendo dentro del umbral aceptable de tolerancia.

Tabla 8 *Adecuación del tiempo de respuesta - Usuario*

ID	Solicitud	Express	Gin	Phoenix	Unidad / Nota
PTb-2-G	POST (obj 1500 ms)	0.282	0.185	0.179	X < 1 se considera bueno.
	GET (obj 800 ms)	0.473	0.298	0.308	
	PUT (obj 1500 ms)	0.294	0.193	0.181	

	DELETE (obj 1000 ms)	0.413	0.286	0.276

En la Figura 4, se puede ver el grafico de la tabla anterior para una mayor comprensión.

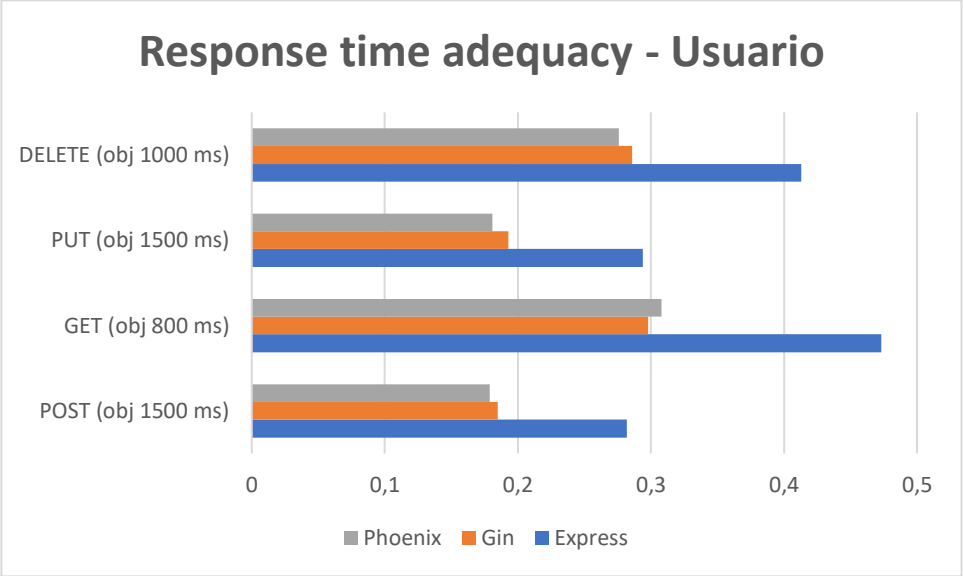


Figura 4 Adecuación del tiempo de respuesta - Usuario

Al evaluar el escenario transaccional complejo del CRUD de Facturas, la introducción del patrón maestro-detalle provocó una degradación generalizada en la métrica, derivando en el incumplimiento de varios de los objetivos preestablecidos. Como se detalla en la **Tabla 9**, el entorno desarrollado con Express sufrió la mayor penalización, invalidando su adecuación en el 100% de las operaciones al superar el umbral permitido ($X > 1$), con un pico crítico de 1.576 en las solicitudes GET.

Por otro lado, Gin y Phoenix mostraron una resiliencia superior en las tareas de escritura, logrando mantenerse en el rango de conformidad técnica ($X < 1$) durante las peticiones POST y PUT; en este aspecto, Phoenix logró el coeficiente más holgado con un 0.715 en la inserción de datos. Sin embargo, ninguna de las tres arquitecturas fue capaz de cumplir con las metas fijadas para las solicitudes GET y DELETE. A pesar de estos incumplimientos parciales en la lectura y eliminación de facturas, Phoenix registró los valores más bajos y cercanos al límite tolerable, superando por muy poco el umbral en la operación DELETE con 1.043.

Tabla 9 *Adecuación del tiempo de respuesta - Factura*

ID	Solicitud	Express	Gin	Phoenix	Unidad / Nota
PTb-2-G	POST (obj 1500 ms)	1.012	0.980	0.715	X < 1 se considera bueno.
	GET (obj 800 ms)	1.576	1.550	1.323	
	PUT (obj 1500 ms)	1.038	0.913	0.865	
	DELETE (obj 1000 ms)	1.450	1.214	1.043	

La Figura 5 ilustra de manera grafica los datos anteriores para una mejor percepción de los resultados.

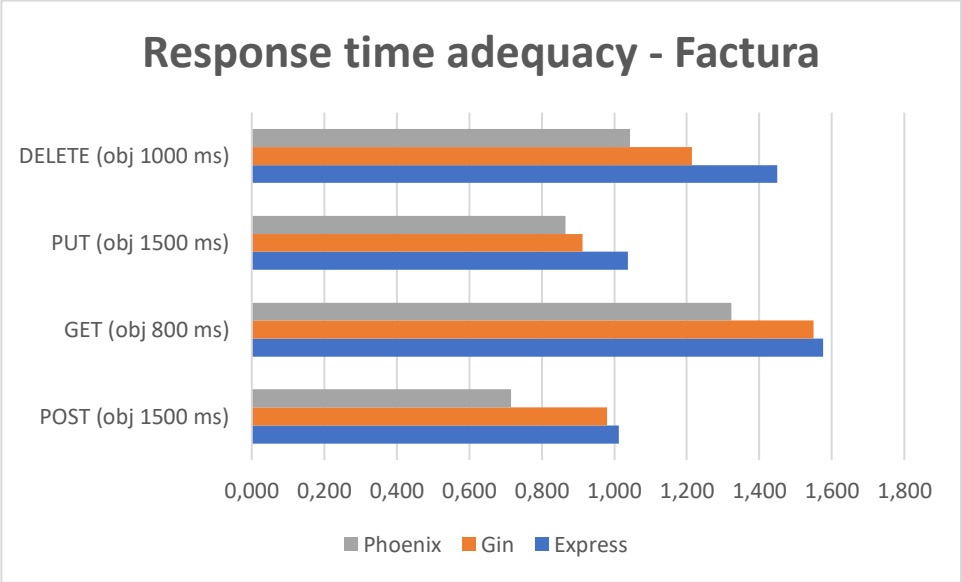


Figura 5 Adecuación del tiempo de respuesta - Factura

4.1.3 Rendimiento medio

Esta métrica de comportamiento temporal mide la cantidad de transacciones completadas por unidad de tiempo, cuantificada a través del volumen total de solicitudes y la tasa de peticiones por segundo (RPS). Como se observa en la **Tabla 10**, en el escenario del CRUD de Usuarios, los frameworks Gin y Phoenix lideran la capacidad de procesamiento con un rendimiento muy similar, registrando 240.14 RPS y 238.98 RPS respectivamente. Por su parte, Express mostró el desempeño menos favorable en este apartado, con una tasa de 229.19 RPS y un menor total de peticiones atendidas durante la prueba.

Tabla 10 *Rendimiento medio - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PTb-5-G	Total de solicitudes	55 806	58 391	58 140	peticiones
	RPS	229.19	240.14	238.98	peticiones/s

Para esclarecer la tendencia de los datos anteriores, la Figura 6 muestra el grafico correspondiente a la tasa de peticiones por segundo.

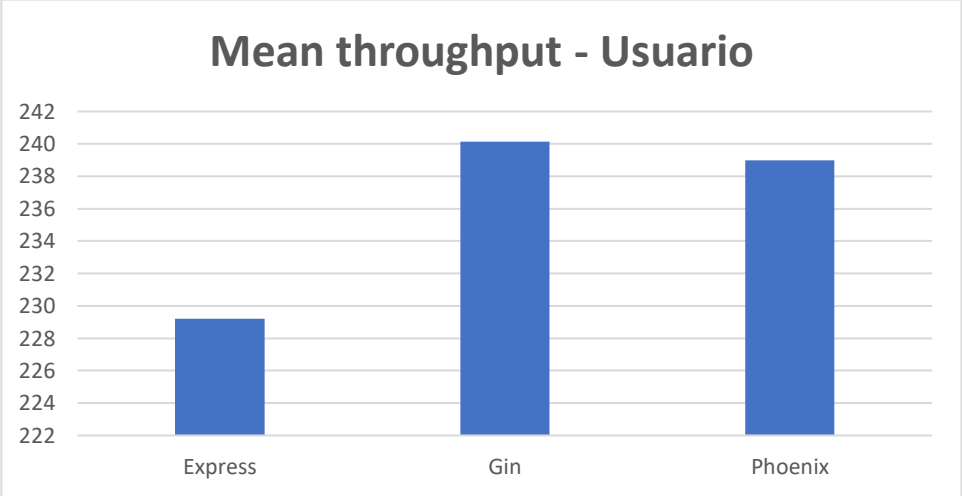


Figura 6 Rendimiento medio - Usuario

Al evaluar el escenario transaccional complejo del CRUD de Facturas, se evidencia una degradación generalizada en el rendimiento medio de todas los frameworks debido al incremento en la carga operativa. Como se detalla en la **Tabla 11**, el ecosistema Phoenix demostró la mayor resiliencia ante este escenario, con una tasa de 167.04 RPS y un total de 40 771 peticiones procesadas. Gin obtuvo estadísticas cercanas y competitivas, consolidando una media de 161.34 RPS. En contraste, Express sufrió la mayor penalización en su eficiencia, reduciendo su capacidad de procesamiento de forma notable hasta los 123.43 RPS, lo que evidencia la vulnerabilidad de su rendimiento ante flujos de datos complejos.

Tabla 11 *Rendimiento medio - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PTb-5-G	Total de solicitudes	30 328	39 371	40 771	peticiones

	RPS	123.43	161.34	167.04	peticiones/s

Los datos obtenidos en RPS se grafican en la Figura 7.

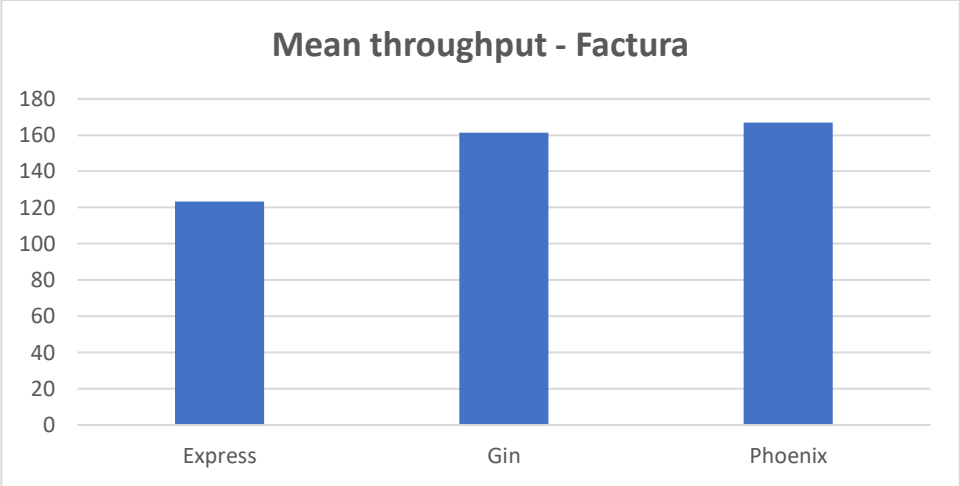


Figura 7 Rendimiento medio - Factura

4.1.4 Utilización media del procesador

Esta métrica de utilización de recursos mide el porcentaje de capacidad de procesamiento del CPU, consumido por cada arquitectura para ejecutar las tareas asignadas en relación con el tiempo total de la prueba, evaluado sobre un límite de entorno rígidamente asignado de 0.5 CPU. Como se observa en la **Tabla 12**, en el escenario del CRUD de Usuarios, el framework Gin demostró la mayor eficiencia en el uso de hardware, registrando el menor consumo con un 21.32% de utilización. Por otro lado, Express y Phoenix mostraron un requerimiento computacional superior y muy similar entre sí, demandando un 33.45% y un 32.56% de la CPU disponible, respectivamente.

Tabla 12 *Utilización media del procesador - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-1-G	CPU (%)	33.45%	21.32%	32.56%	CPU disponible: 0.5

Para una interpretación rápida, la Figura 8 muestra el porcentaje de CPU utilizado.

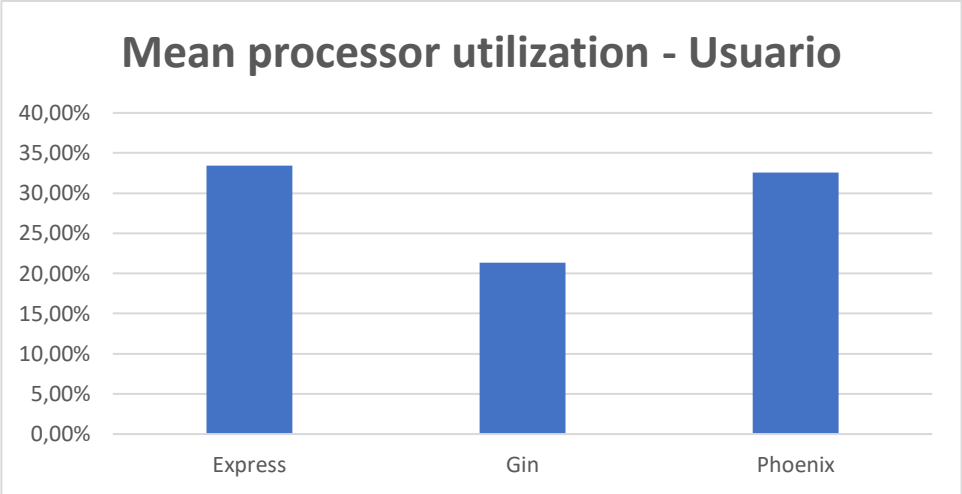


Figura 8 Utilización media del procesador - Usuario

Al evaluar la lógica maestro-detalle se identificó un cambio ligero, reflejando cómo cada entorno gestiona los recursos de cómputo frente a la carga transaccional. Como se detalla en la **Tabla 13**, el ecosistema Phoenix registró la mayor utilización de procesamiento en este escenario con un 30.27%, seguido muy de cerca por Express con un 29.88%. Por otro lado, el framework Gin se mantuvo nuevamente como la arquitectura más eficiente en términos de cómputo puro, requiriendo apenas un 23.54% de la CPU, logrando el menor impacto sobre el procesador a pesar del incremento en la complejidad de las operaciones de persistencia.

Tabla 13 *Utilización media del procesador - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-1-G	CPU (%)	29.88%	23.54%	30.27%	CPU disponible: 0.5

El porcentaje de CPU usado se muestra, a continuación, en la Figura 9.

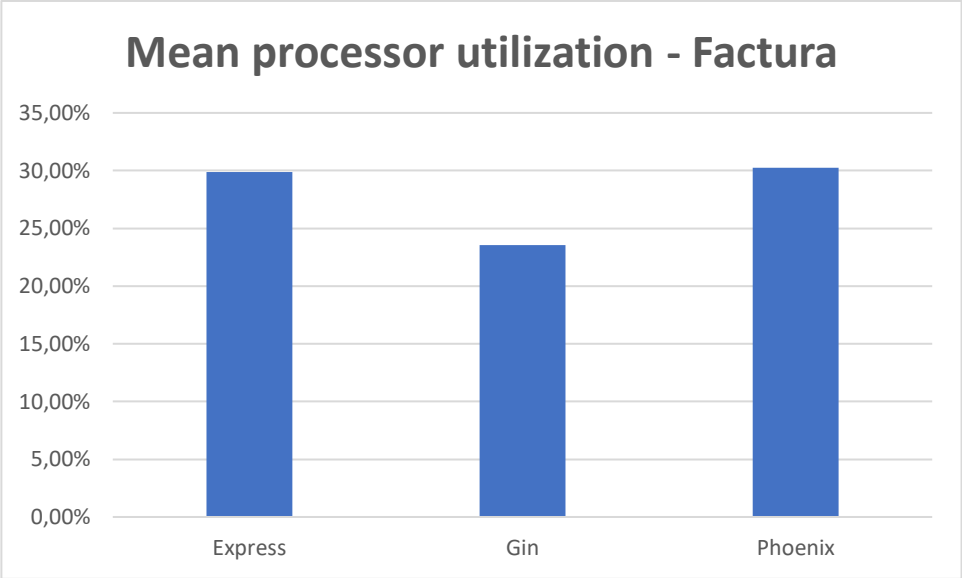


Figura 9 Utilización media del procesador - Factura

4.1.5 Utilización media de memoria

Esta métrica de utilización de recursos determina la cantidad de memoria RAM consumida por cada arquitectura en relación con la memoria total disponible en el sistema, evaluada sobre un límite rígidamente asignado de 512 MB. Como se observa en la **Tabla 14**, en el escenario del CRUD de Usuarios, el framework Gin demostró una notable eficiencia en la gestión de hardware, registrando el menor consumo con apenas un 3.23% de utilización (16.6 MB). Por su parte, Express se posicionó en un rango intermedio con un 7.13% (36.5 MB), mientras que Phoenix mostró el consumo de memoria más elevado en este apartado, demandando un 33.68% de la capacidad total asignada (172.4 MB).

Tabla 14 *Utilización media de memoria - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-2-G	Memoria (%)	7.13%	3.23%	33.68%	Memoria disponible: 512 MB
	Memoria	36.5	16.6	172.4	MB

En la Figura 10 se puede observar de manera grafica el porcentaje de memoria utilizado por cada framework.

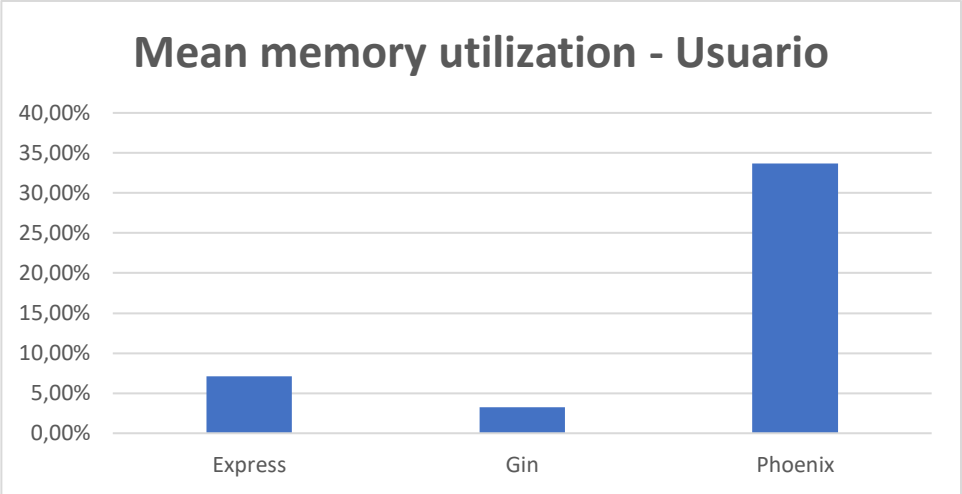


Figura 10 Utilización media de memoria - Usuario

Al evaluar el escenario transaccional complejo, este provocó un incremento generalizado en la retención de memoria en los tres frameworks debido a la carga y manipulación de estructuras de datos simultáneas en la memoria intermedia. Como se detalla en la **Tabla 15**, el framework Gin se mantuvo nuevamente como la opción óptima en el uso de este recurso, elevando su consumo a solo un 4.85% (24.8 MB). Express registró un comportamiento controlado con un 8.30% de utilización (42.5 MB), mientras que el ecosistema Phoenix requirió la mayor asignación de hardware, alcanzando un 41.53% de consumo de memoria RAM (212.6 MB) ante flujos de persistencia de datos complejos.

Tabla 15 *Utilización media de memoria - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-2-G	Memoria (%)	8.30%	4.85%	41.53%	Memoria disponible: 512 MB
	Memoria	42.5	24.8	212.6	MB

Para un entendimiento mayor, en la Figura 11 se grafica el porcentaje de memoria utilizado.

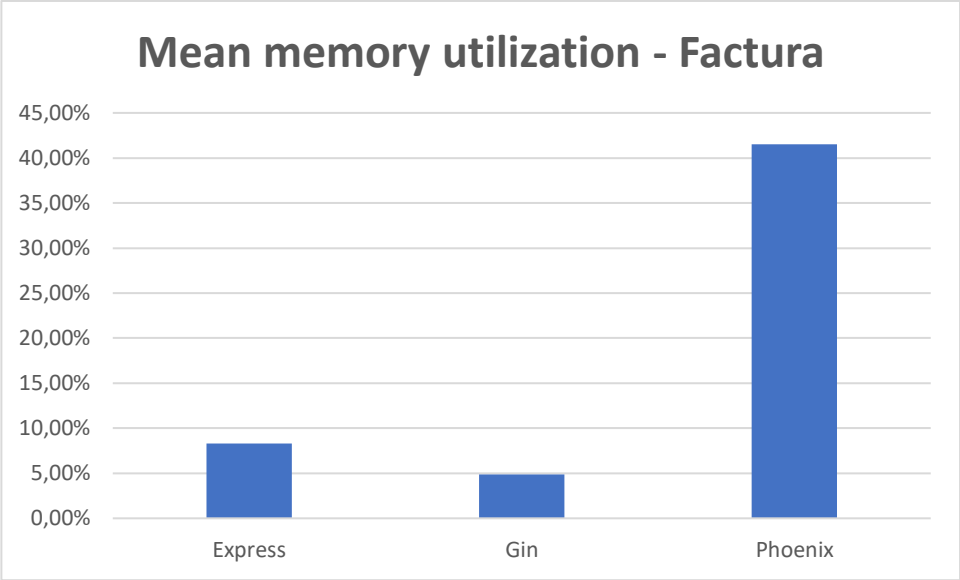


Figura 11 Utilización media de memoria - Factura

4.1.6 Utilización del ancho de banda

Esta métrica de utilización de recursos mide la tasa del ancho de banda de red ocupada durante la ejecución de las tareas, evaluada a través de la tasa de envío de datos y de recepción de datos en bits por segundo (bps). Como se observa en la **Tabla 16**, en el escenario del CRUD de Usuarios, las tres arquitecturas muestran un comportamiento sumamente equilibrado debido a la simplicidad del modelo de datos transferido. El ecosistema Phoenix registró la mayor tasa de recepción con 60 848 bps, seguido por Express con 58 009 bps y Gin con 55 319 bps. En cuanto al envío de información, Gin y Phoenix presentaron transferencias ligeramente superiores a Express, situándose en 21 751 bps y 21 641 bps respectivamente.

Tabla 16 *Utilización del ancho de banda - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-4-S	Tasa de envío de datos	20 771	21 751	21 641	bits/s
	Tasa de recepción de datos	58 009	55 319	60 848	bits/s

En la Figura 12 se presenta de manera grafica los datos anteriores para una mayor percepción.

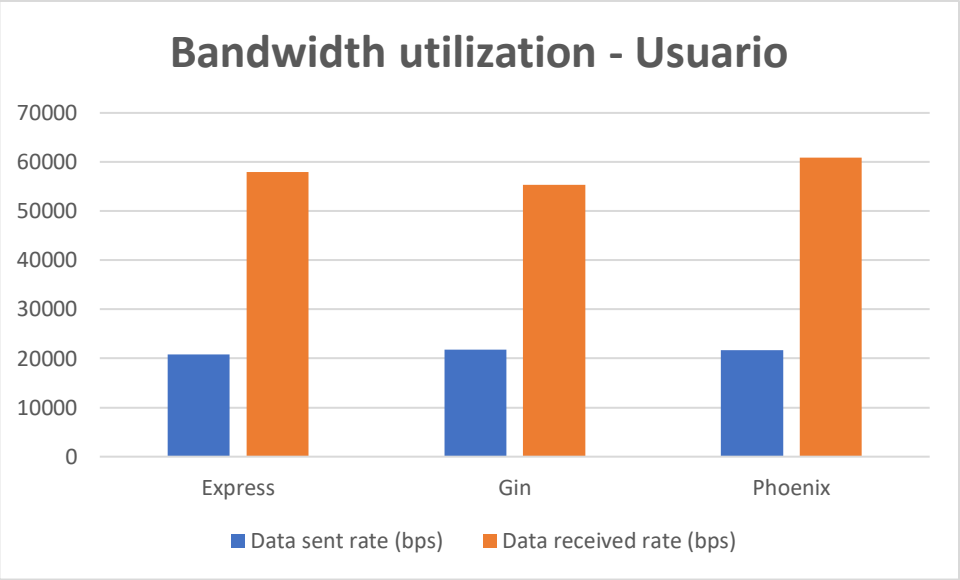


Figura 12 Utilización del ancho de banda - Usuario

Al evaluar la estructura maestro-detalle se evidenció un cambio notable en el tráfico de red de los entornos, reflejando el incremento en el tamaño de los mensajes y el volumen de solicitudes procesadas. Como se detalla en la **Tabla 17**, el framework Phoenix experimentó un notable incremento en su tasa de recepción de datos, alcanzando los 153 870 bps, manteniéndose también al frente en el envío de información con 24 601 bps. Por su parte, Gin registró métricas intermedias con una recepción de 54 731 bps y un envío de 23 126 bps. En contraste, Express mostró las tasas de transferencia más bajas del experimento, disminuyendo su recepción a 45 914 bps y su envío a 18 017 bps, lo que guarda relación directa con su menor tasa de procesamiento global ante operaciones complejas.

Tabla 17 *Utilización del ancho de banda - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-4-S	Tasa de envío de datos	18 017	23 126	24 601	bits/s
	Tasa de recepción de datos	45 914	54 731	153 870	bits/s

En la Figura 13 se grafican los datos anteriores para una mayor comprensión.

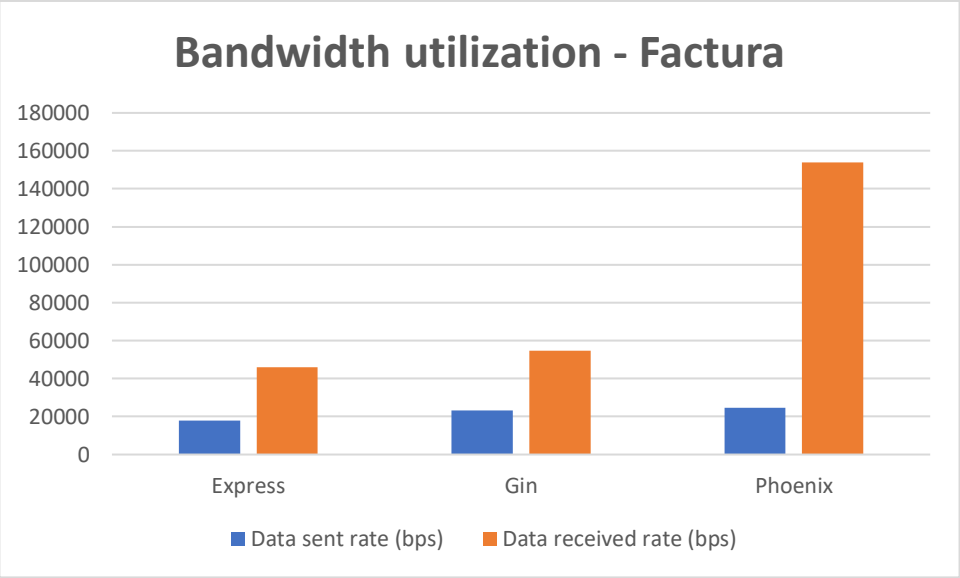


Figura 13 Utilización del ancho de banda - Factura

4.1.7 Capacidad de procesamiento de transacciones

Esta métrica de capacidad determina el número máximo de solicitudes totales que el sistema puede procesar por unidad de tiempo, bajo una carga extrema o prueba de estrés, antes de alcanzar su punto de quiebre. Como se observa en la **Tabla 18**, en el escenario del CRUD de Usuarios, el ecosistema Phoenix demostró la mayor capacidad de escalabilidad al tolerar 2 173 VUs concurrentes y evacuar un volumen masivo de 275 392 peticiones totales. Por su parte, Express registró un punto de quiebre intermedio en torno a los 1 711 VUs. En tanto, el framework Gin alcanzó su límite de forma más temprana con aproximadamente 1 448 VUs; no obstante, cabe destacar que la arquitectura de Go fue la única con una tasa de error aproximadamente del 0% al momento del colapso, el cual fue provocado en los tres entornos por la violación del umbral crítico de latencia (P95 superior a 5 segundos).

Tabla 18 *Capacidad de procesamiento de transacciones - Usuario*

ID	Parámetro	Express	Gin	Phoenix	Unidad / Nota
PCa-1-G	Usuarios en el punto de quiebre	1 711	1448	2 173	VUs
	Total de solicitudes hasta quiebre	136 232	148 523	275 392	peticiones

	Tasa de error en quiebre	0.52%	≈0%	6.70%	umbral establecido <10%
	Causa del aborto	Latencia P95 >5s	Latencia P95 >5s	Latencia P95 >5s	

El total de solicitudes se representa gráficamente, en la Figura 14, para una mayor comprensión.

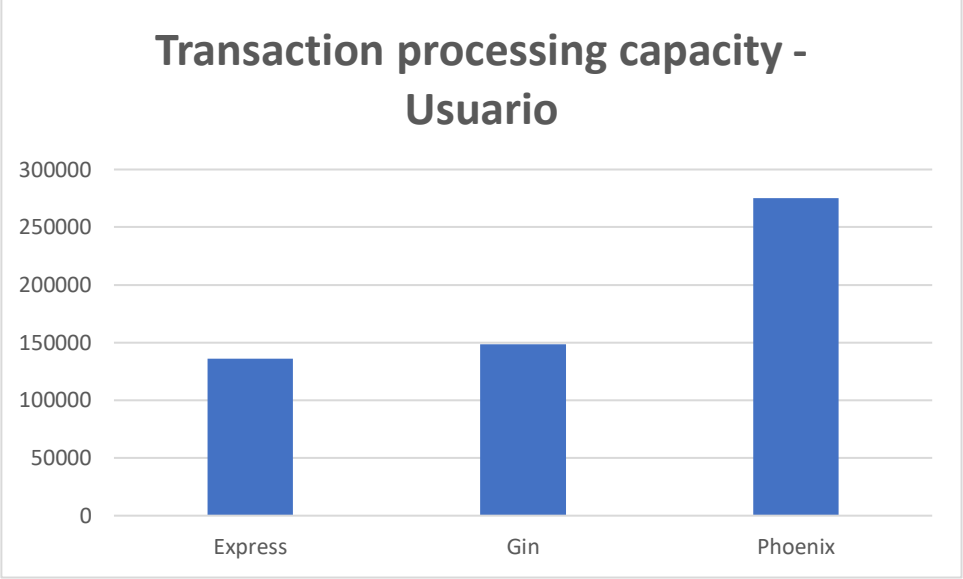


Figura 14 Capacidad de procesamiento de transacciones - Usuario

Al evaluar el escenario transaccional complejo del CRUD de Facturas, se redujo drásticamente el techo de resistencia de todos los entornos debido a la saturación concurrente en la persistencia de datos. Como se detalla en la **Tabla 19**, el comportamiento de las arquitecturas varió en este escenario. Express demostró la mayor tolerancia cuantitativa al soportar alrededor de 718 VUs concurrentes antes de degradar su latencia más allá de los 5 segundos. Gin dejó ver un comportamiento robusto al resistir aproximadamente 574 VUs manteniendo un 0% de errores en las transacciones antes de colapsar por tiempo de respuesta.

Por el contrario, el ecosistema Phoenix sufrió la penalización más severa bajo este escenario de estrés complejo, alcanzando prematuramente su punto de quiebre con apenas 259 VUs concurrentes. A diferencia de los demás frameworks, la causa del aborto en Phoenix no estuvo ligada a la degradación de la latencia, sino a la superación del límite de tolerancia de fallos del sistema, registrando una tasa de error del 11.25% (sobrepasando

el umbral máximo permitido del 10%), lo que evidencia una vulnerabilidad estructural en su persistencia transaccional masiva bajo condiciones de fatiga extrema.

Tabla 19 *Capacidad de procesamiento de transacciones - Factura*

ID	Parámetro	Express	Gin	Phoenix	Unidad / Nota
PCa-1-G	Usuarios en el punto de quiebre	718	574	259	VUs
	Total de solicitudes hasta quiebre	23 600	22 744	9 943	peticiones
	Tasa de error en quiebre	0.63%	0%	11.25%	umbral establecido <10%
	Causa del aborto	Latencia P95 > 5s	Latencia P95 > 5s	Tasa de error >10%	

En la Figura 15 se presenta de igual forma el total de solicitudes, para así lograr un mayor entendimiento.

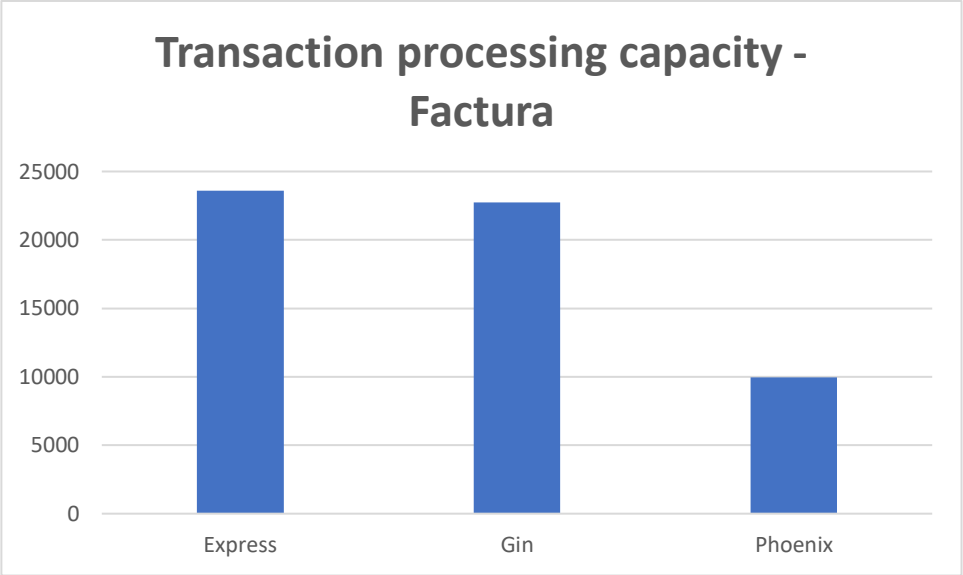


Figura 15 Capacidad de procesamiento de transacciones - Factura

4.1.8 Capacidad de acceso de usuarios

Esta métrica de capacidad, indica el número máximo de VUs concurrentes que pueden acceder simultáneamente al sistema, manteniendo una tasa de fallo controlada por debajo del umbral del 10%, cuantificando además el volumen total de operaciones exitosas alcanzadas hasta ese límite. Como se observa en la **Tabla 20**, bajo el escenario simple del CRUD de Usuarios, el ecosistema Phoenix demostró la mayor capacidad de soporte simultáneo al registrar un límite de aproximadamente 2 173 usuarios concurrentes y consolidar 256 920 operaciones exitosas. Express se situó en un rango intermedio con alrededor de 1 711 usuarios, mientras que Gin registró el techo de acceso más temprano con cerca de 1 448 usuarios concurrentes, aunque logrando un total competitivo de 148 523 operaciones completadas con éxito.

Tabla 20 *Capacidad de acceso de usuarios - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PCa-2-G	Capacidad de acceso de usuarios	1 711	1 448	2 173	VUs
	Operaciones exitosas	135 523	148 523	256 920	peticiones

Los usuarios máximos alcanzados son graficados en la Figura 16, para su compresión.

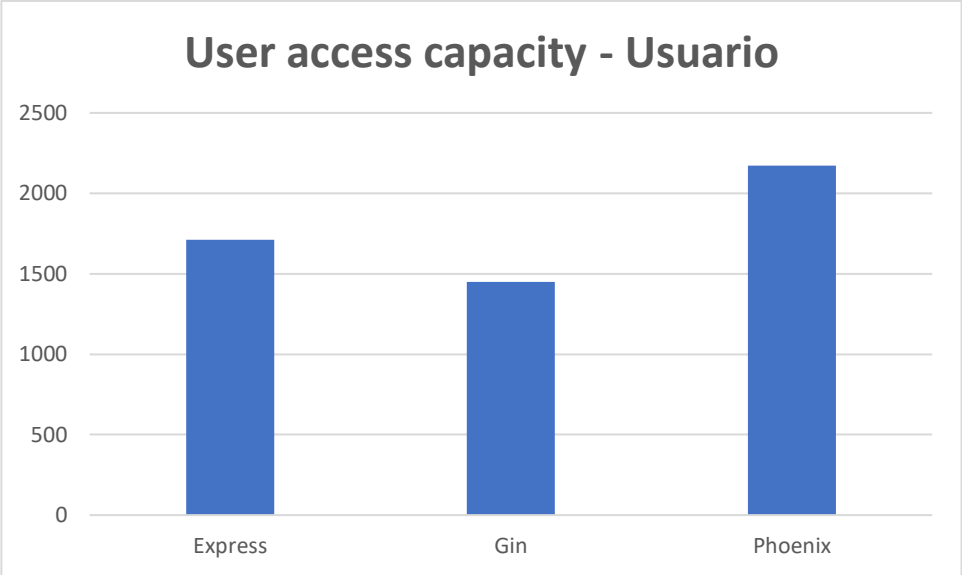


Figura 16 Capacidad de acceso de usuarios - Usuario

Al evaluar el escenario transaccional complejo del CRUD de Facturas, se redujo notablemente los límites de accesos simultáneos seguros en todos los entornos. Como se detalla en la **Tabla 21**, el comportamiento de las arquitecturas varió de manera inversa al escenario simple. El framework Express demostró la mayor resiliencia cuantitativa en este apartado, manteniendo su tasa de error por debajo del límite estipulado hasta alcanzar un aproximado de 718 usuarios concurrentes y 23 450 operaciones exitosas. Gin se mantuvo en una posición intermedia al tolerar cerca de 574 usuarios simultáneos. En contraposición, el entorno desarrollado en Phoenix experimentó la contracción más severa frente a la carga compleja, viendo restringida su capacidad de acceso a únicamente alrededor de 259 usuarios antes de que su tasa de errores sobrepasara el límite permisible del 10%.

Tabla 21 *Capacidad de acceso de usuarios - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PCa-2-G	Capacidad de acceso de usuarios	718	574	259	VUs
	Operaciones exitosas	23 450	22 744	8 831	peticiones

La Figura 17 , muestra gráficamente la cantidad máxima de usuarios alcanzados en los datos anteriores.

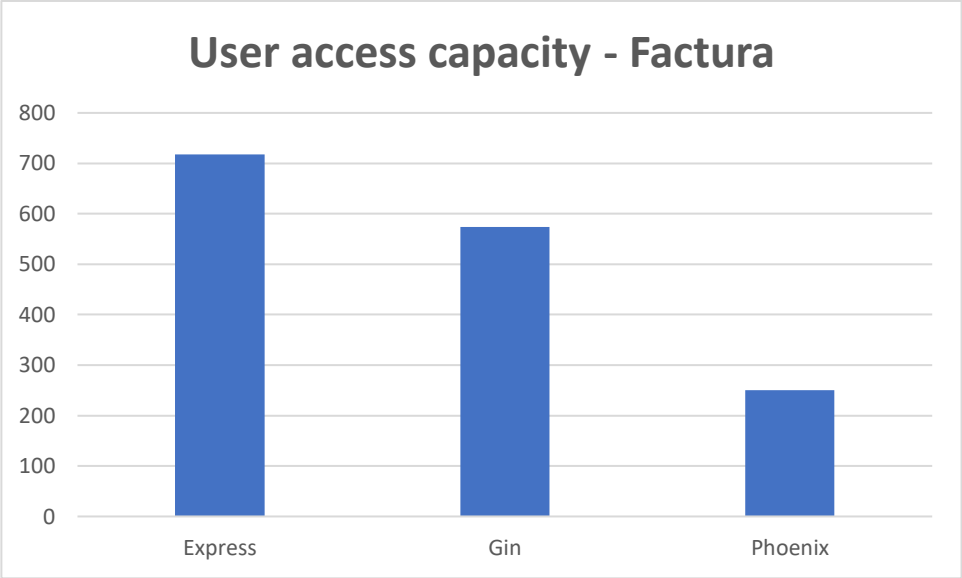


Figura 17 Capacidad de acceso de usuarios - Factura

4.1.9 Adecuación al incremento del acceso de usuarios

Esta métrica de capacidad evalúa el grado de adecuación del sistema para absorber nuevos usuarios concurrentes en escenarios de crecimiento rápido, calculado como un coeficiente adimensional que contrasta los VUs máximos alcanzados de forma segura frente a un objetivo de escalabilidad ideal establecido de 3 000 VUs. Como se observa en la **Tabla 22**, en el escenario del CRUD de Usuarios, el ecosistema Phoenix demostró la mejor respuesta de escalabilidad vertical al consolidar el índice más alto con un 0.724 de adecuación. El framework Express se posicionó en un rango intermedio registrando un factor de 0.570, mientras que Gin exhibió el menor valor ante un incremento rápido de accesos en este entorno simple, situándose en un 0.483.

Tabla 22 *Adecuación al incremento del acceso de usuarios - Usuario*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PCa-3-S	Adecuación al incremento del acceso de usuarios	0.570	0.483	0.724	Objetivo establecido: 3000 VUs

Para comprender de mejor manera los datos, los datos anteriores fueron graficados en la Figura 18.

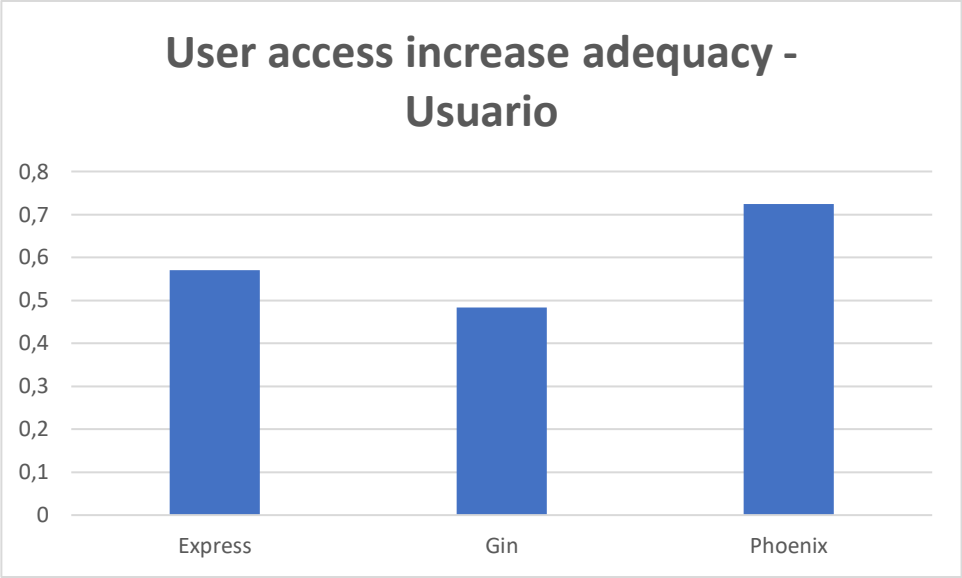


Figura 18 Adecuación al incremento del acceso de usuarios - Usuario

Al evaluar la lógica maestro-detalle, se pudo observar que provocó una contracción severa en los coeficientes de adecuación de todas las arquitecturas, reflejando el impacto que sufre el escalado rápido cuando el sistema se ve condicionado por la capa de persistencia. Como se detalla en la **Tabla 23**, el framework Express retuvo la mayor resiliencia proporcional en este entorno extremo con un índice de 0.239, seguido por Gin con un factor de 0.191. En contraposición, el ecosistema Phoenix sufrió la penalización más crítica de todo el experimento, disminuyendo drásticamente su coeficiente de adecuación a un 0.086, lo que confirma las limitaciones de su infraestructura para asimilar flujos acelerados de nuevos usuarios concurrentes bajo operaciones de persistencia de alta complejidad transaccional.

Tabla 23 *Adecuación al incremento del acceso de usuarios - Factura*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PCa-3-S	Adecuación al incremento del acceso de usuarios	0.239	0.191	0.086	Objetivo establecido: 3000 VUs

Por último, en la Figura 19 se ve de manera grafica los datos de la tabla anterior.

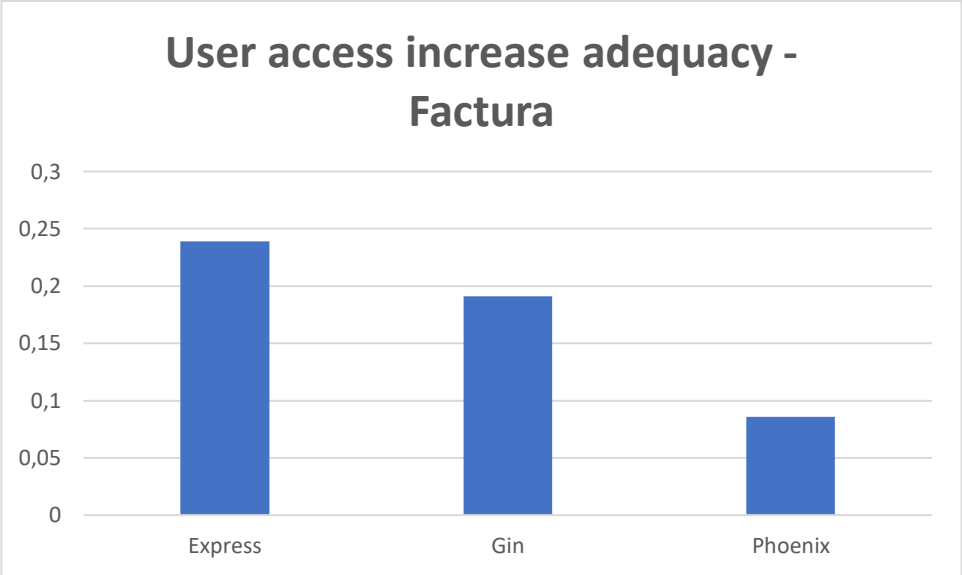


Figura 19 Adecuación al incremento del acceso de usuarios - Factura

4.2 Arquitectura de Microservicios

Al introducir la capa de abstracción y enrutamiento del API Gateway, se observa una homogenización en los tiempos de respuesta debido a la latencia base del proxy asíncrono, reduciendo las brechas operativas observadas en el entorno monolítico.

4.2.1 Tiempo medio de respuesta

Como se detalla en la **Tabla 24**, Phoenix demostró la mayor eficiencia adaptativa dentro de esta infraestructura distribuida, liderando la prueba con un tiempo promedio global de 209.07 ms y manteniendo una notable estabilidad en todas sus operaciones CRUD. Por su parte, Express se posicionó en un rango intermedio muy competitivo, registrando una media total de 215.79 ms. En contraposición, el entorno desarrollado en Gin mostró el desempeño menos favorable bajo esta arquitectura, elevando su tiempo promedio general a 224.51 ms.

El análisis del P95 ratifica la jerarquía de rendimiento de los entornos en este esquema de microservicios. Phoenix consolidó los valores de latencia más bajos y estables, conteniendo todas sus operaciones por debajo de los 274 ms. Express evidenció una dispersión controlada pero superior, con picos de P95 que alcanzaron los 295.53 ms en la inserción de datos (POST). Finalmente, Gin reflejó la mayor fluctuación temporal del experimento, superando la barrera de los 300 ms en las solicitudes de lectura (GET), lo que pone de manifiesto el impacto que sufren los tiempos de servicio al interactuar a través del puente de comunicación asíncrono provisto por el API Gateway.

Tabla 24 *Tiempo medio de respuesta en Microservicios*

ID	Solicitud	Express	Gin	Phoenix	Unidad / Nota
PTb-1-G	POST_avg	231.81	234.90	219.96	ms
	GET_avg	225.01	237.80	219.88	ms
	PUT_avg	225.79	235.81	218.11	ms
	DELETE_avg	224.76	233.65	218.36	ms
	TOTAL_avg	215.79	224.51	209.07	ms
	POST_P95	295.53	288.65	273.32	ms
	GET_P95	280.15	311.18	271.88	ms
	PUT_P95	279.81	299.87	267.54	ms
	DELETE_P95	285.38	291.94	272.15	ms

Para una mejor comprensión, en la Figura 20 se presenta el promedio total y los P95.

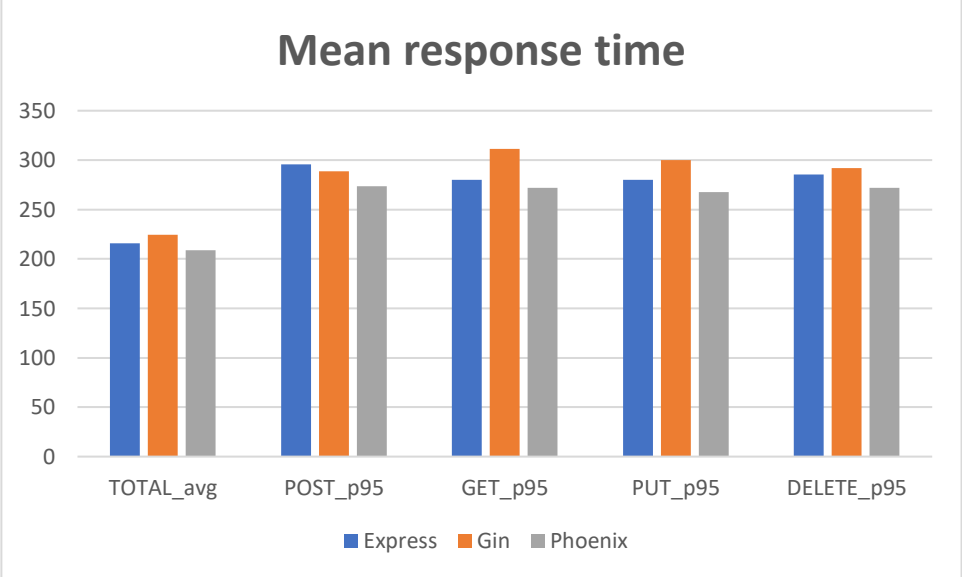


Figura 20 Tiempo medio de respuesta en Microservicios

4.2.2 Adecuación del tiempo de respuesta

Al analizar esta métrica frente a los objetivos establecidos en este entorno descentralizado, se evidencia que la introducción del API Gateway permitió a las tres tecnologías asimilar la carga de forma uniforme. Como se observa en la **Tabla 25**, bajo una carga constante de 25 VUs, los tres entornos de desarrollo cumplieron con éxito y de manera holgada la totalidad de las metas fijadas.

El ecosistema Phoenix consolidó nuevamente los coeficientes de adecuación óptimos en todas las operaciones del CRUD, demostrando una excelente adaptabilidad en el entorno de microservicios y alcanzando su punto más eficiente en las solicitudes PUT con un índice de 0.178. Por su parte, Express y Gin mantuvieron un rendimiento altamente competitivo y cercano entre sí; sin embargo, de acuerdo con la métrica anterior, Gin reflejó una sutil penalización frente a Express en la mayoría de las operaciones, registrando su coeficiente menos favorable en la lectura de datos (GET) con 0.389 ms, mientras que Express se desempeñó con un valor de 0.350.

Tabla 25 *Adecuación del tiempo de respuesta en Microservicios*

ID	**Solicitud**	**Express**	**Gin**	**Phoenix**	**Unidad / Nota**
PTb-2-G	POST (obj 1500 ms)	0.197	0.192	0.182	X < 1 se considera bueno.
	GET (obj 800 ms)	0.350	0.389	0.340	
	PUT (obj 1500 ms)	0.187	0.200	0.178	
	DELETE (obj 1000 ms)	0.285	0.292	0.272	

La Figura 21 muestra de manera grafica los datos de la tabla anterior para un mejor entendimiento.

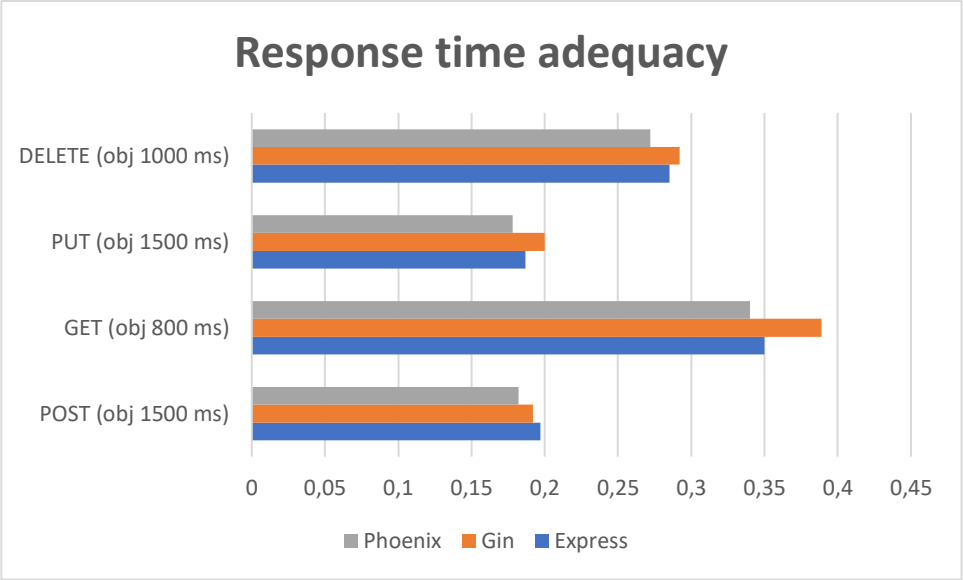


Figura 21 Adecuación del tiempo de respuesta en Microservicios

4.2.3 Rendimiento medio

En la evaluación de la capacidad de procesamiento bajo el esquema descentralizado, los resultados consolidados en la **Tabla 26** reflejan una notable estabilización y paridad en la tasa de transferencia de las tres tecnologías. El ecosistema Phoenix registró el rendimiento medio más alto con una tasa de 39.46 RPS y un acumulado de 9 622 peticiones totales atendidas. Express se ubicó en una posición intermedia muy cercana, con 39.08 RPS, mientras que Gin obtuvo el indicador sutilmente menor con 38.57 RPS.

Tabla 26 *Rendimiento medio en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PTb-5-G	Total de solicitudes	9 514	9 390	9 622	peticiones
	RPS	39.08	38.57	39.46	peticiones/s

Los datos de RPS se muestran gráficamente en la Figura 22.

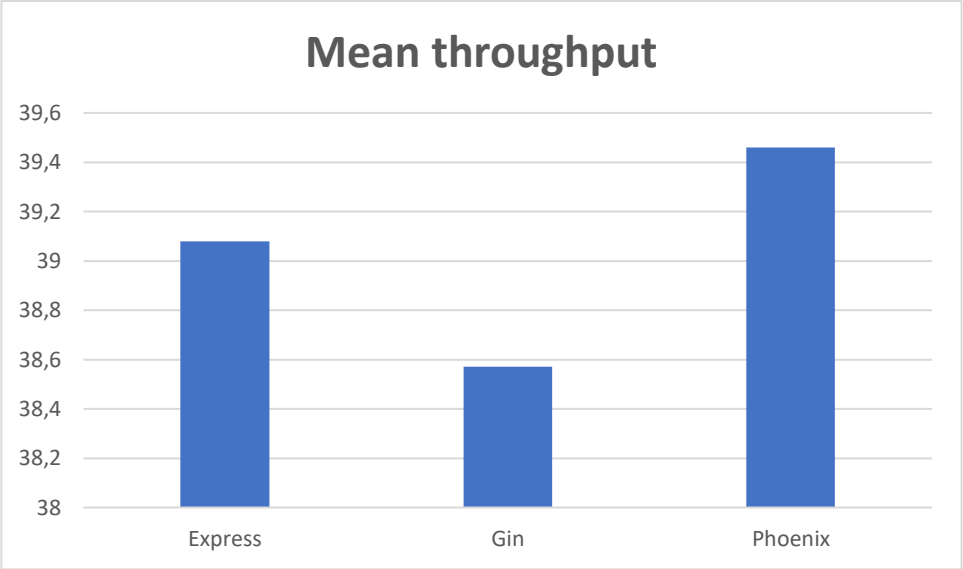


Figura 22 Rendimiento medio en Microservicios

4.2.4 Utilización media del procesador

En la evaluación del consumo de recursos computacionales dentro del entorno distribuido, los datos registrados en la **Tabla 27** revelan un uso moderado y equilibrado entre las tres tecnologías, operando todas de forma holgada por debajo del límite asignado de 0.5 CPU. El framework Gin revalidó su alta eficiencia en el uso de hardware al registrar la menor demanda sobre el procesador, requiriendo apenas un 22.01% de utilización.

Por su parte, Phoenix y Express exhibieron un comportamiento prácticamente idéntico, demandando un 25.94% y un 25.70% de la CPU disponible, respectivamente.

Tabla 27 *Utilización media del procesador en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-1-G	CPU (%)	25.70%	22.01%	25.94%	CPU disponible: 0.5

Para esclarecer los datos anteriores, la Figura 23 muestra el porcentaje de CPU utilizado.

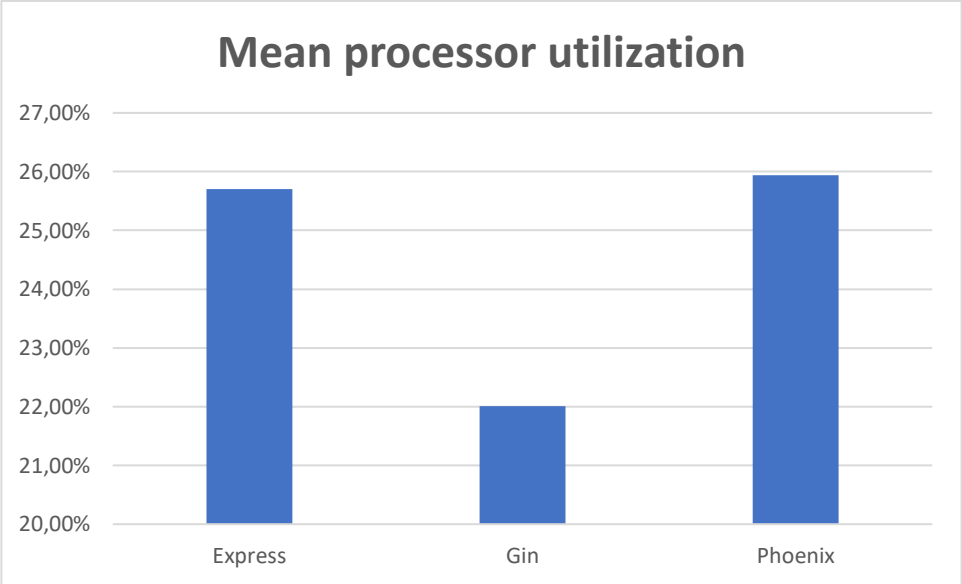


Figura 23 Utilización media del procesador en Microservicios

4.2.5 Utilización media de memoria

En la evaluación del consumo de memoria RAM dentro del ecosistema distribuido, los datos recopilados en la **Tabla 28** revelan un fenómeno de homogeneización casi absoluta entre las tres tecnologías, operando todas en un rango estrecho entre el 22% y el 23% del límite asignado de 512 MB. A diferencia del escenario monolítico previo, donde las huellas de memoria nativas de cada framework marcaban brechas notables, en este entorno Express registró un consumo del 22.23% (113.8 MB), Phoenix un 22.27% (114.0 MB) y Gin un 23.07% (118.1 MB).

Tabla 28 *Utilización media de memoria en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-2-G	Memoria (%)	22.23%	23.07%	22.27%	Memoria disponible: 512 MB
	Memoria	113.8	118.1	114.0	MB

La Figura 24 muestra de manera grafica los datos del porcentaje de CPU utilizado, para una mejor percepción de los resultados.

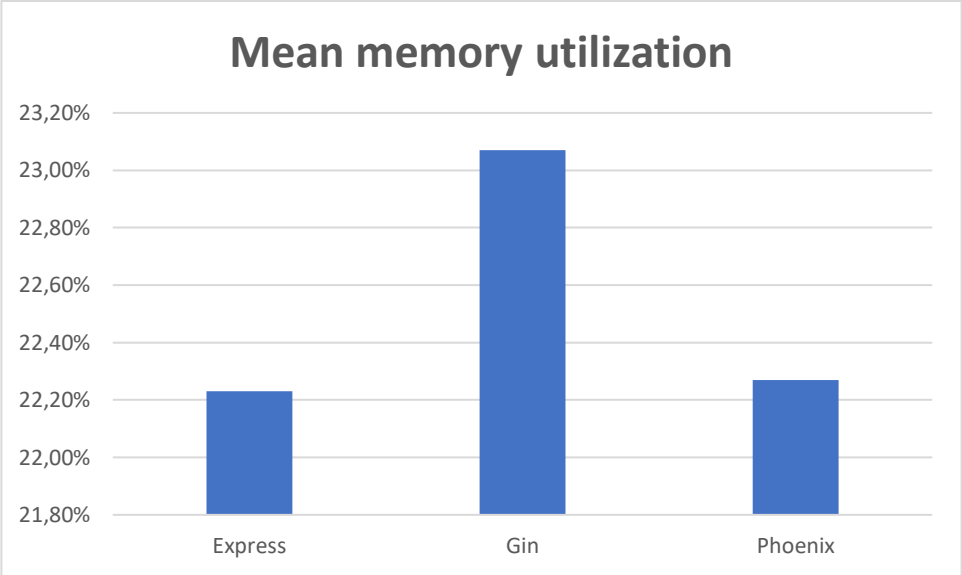


Figura 24 Utilización media de memoria en Microservicios

4.2.6 Utilización del ancho de banda

En la evaluación del consumo de red dentro del esquema distribuido, los resultados de la **Tabla 29** reflejan una correlación directa y proporcional con el rendimiento medio registrado previamente por cada tecnología. La tasa de transferencia de datos se mantuvo uniforme entre las tres arquitecturas debido a la estandarización de los payloads JSON que entran y salen a través del API Gateway.

Phoenix registró las tasas de transferencia ligeramente más altas, con una tasa de recepción de datos de 9 237 bps y un envío de datos de 4 585 bps, lo cual es consecuente con su mayor volumen de peticiones atendidas por segundo. Express se posicionó nuevamente en el rango intermedio con una recepción de 9 079 bps y un envío de 4 539 bps. Por último, Gin exhibió el menor impacto sobre el ancho de banda, con una tasa de recepción de 8 412 bps y un envío de 4 481 bps.

Tabla 29 *Utilización del ancho de banda en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PRu-4-S	Tasa de envío de datos	4 539	4 481	4 585	bits/s
	Tasa de recepción de datos	9 079	8 412	9 237	bits/s

La Figura 25 muestra los datos anteriores gráficamente, para una mejor comprensión.

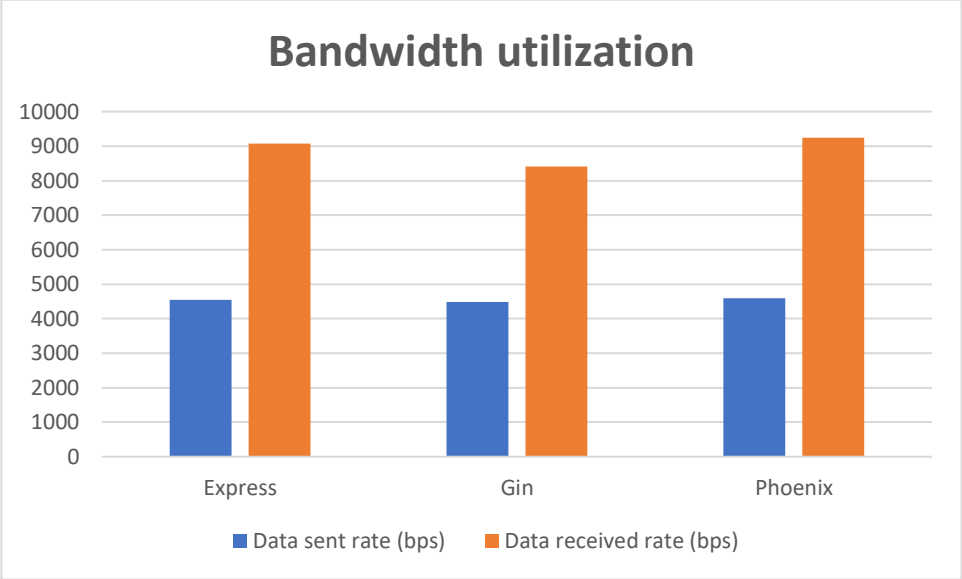


Figura 25 Utilización del ancho de banda en Microservicios

4.2.7 Capacidad de procesamiento de transacciones

En la evaluación de la capacidad bajo condiciones extremas, los resultados muestran un comportamiento de saturación altamente unificado. A diferencia del entorno monolítico, la transición a una arquitectura descentralizada provocó que las tres tecnologías alcanzaran su punto de quiebre en un umbral muy estrecho y bajo la misma causa de aborto: la degradación del tiempo de respuesta, manteniendo tasas de error aproximadamente inexistentes.

Como se detalla en la **Tabla 30**, Gin y Phoenix exhibieron un rendimiento casi idéntico en su límite de resistencia, tolerando un máximo de 433 VUs (22 128 peticiones) y 431 VUs (22 018 peticiones) respectivamente antes del colapso de latencia. Por su parte, Express registró una saturación sutilmente más temprana, situando su punto de quiebre en torno a las 410 VUs concurrentes y realizando 19 347 solicitudes.

El hallazgo más relevante de este análisis radica en la contracción del techo de escalabilidad de los frameworks en comparación con sus resultados monolíticos previos.

Tabla 30 *Capacidad de procesamiento de transacciones en Microservicios*

ID	Parámetro	Express	Gin	Phoenix	Unidad / Nota

PCa-1-G	Usuarios en el punto de quiebre	410	433	431	VUs
	Total de solicitudes hasta quiebre	19 347	22 128	22 018	peticiones
	Tasa de error en quiebre	0.035%	0%	0%	umbral establecido <10%
	Causa del aborto	Latencia P95 >5s	Latencia P95 >5s	Latencia P95 >5s	

Para un entendimiento más completo, la Figura 26 presenta el total de solicitudes en un gráfico.

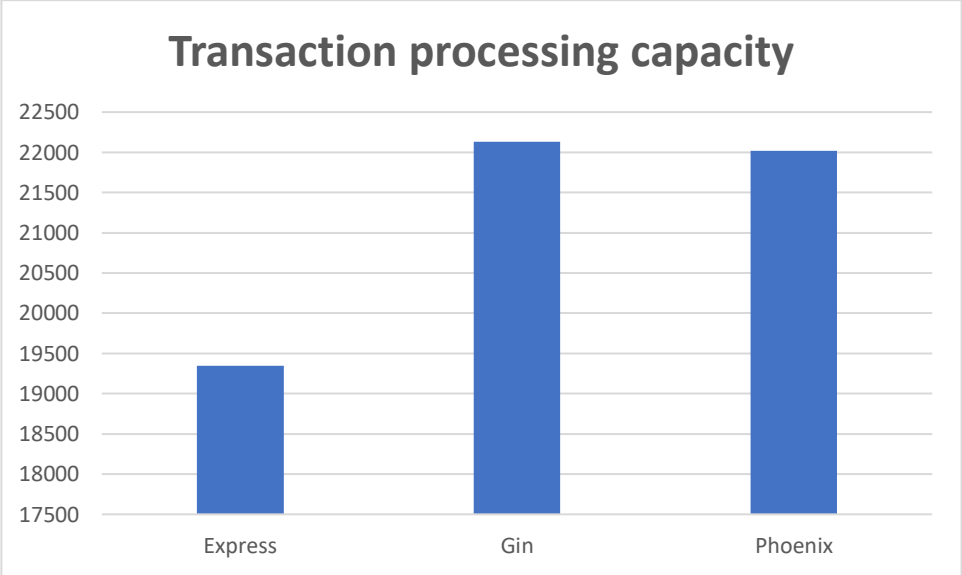


Figura 26 Capacidad de procesamiento de transacciones en Microservicios

4.2.8 Capacidad de acceso de usuarios

Como se detalla en la **Tabla 31**, Gin y Phoenix lideraron este apartado con una paridad técnica casi absoluta, tolerando un volumen máximo de 433 y 431 usuarios concurrentes, respectivamente. En cuanto a productividad neta, la arquitectura en Go consolidó 14,415 operaciones exitosas, mientras que el entorno basado en Elixir alcanzó las 14,340. Por su parte, Express registró la menor tolerancia del experimento al situar su

capacidad límite en aproximadamente 410 usuarios concurrentes, acumulando un total de 12,584 operaciones procesadas con éxito antes de colapsar.

Tabla 31 *Capacidad de acceso de usuarios en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota
PCa-2-G	Capacidad de acceso de usuarios	410	433	431	VUs
	Operaciones exitosas	12,584	14,415	14,340	peticiones

A continuación, se presenta el máximo de usuarios alcanzados en la Figura 27.

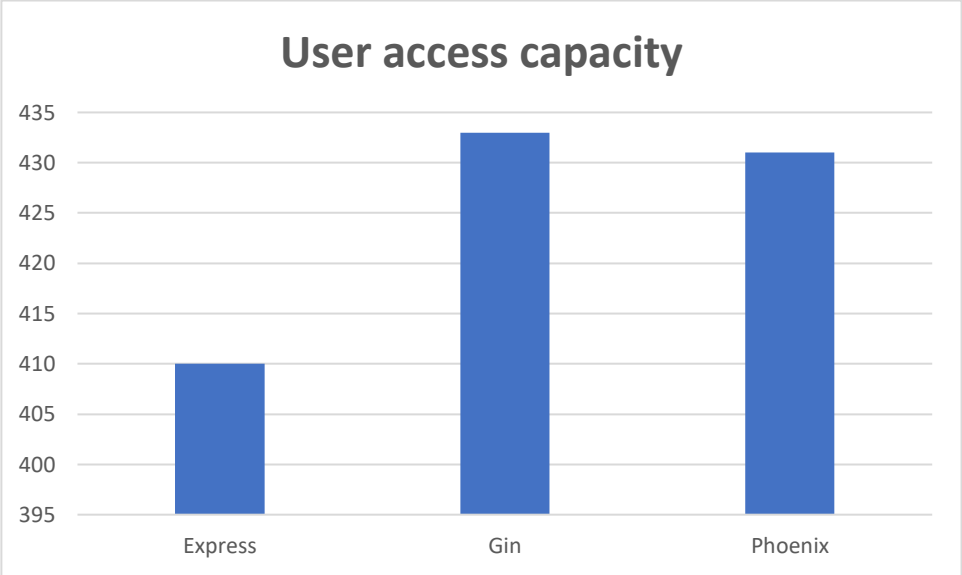


Figura 27 Capacidad de acceso de usuarios en Microservicios

4.2.9 Adecuación al incremento del acceso de usuarios

Como se detalla en la **Tabla 32**, Gin consolidó el coeficiente ligeramente más alto con un valor de 0.577, seguido de forma casi imperceptible por el ecosistema Phoenix con un factor de 0.575. Por su parte, Express registró la menor holgura ante la aceleración de accesos simultáneos, situándose con un índice de 0.547.

Tabla 32 *Adecuación al incremento del acceso de usuarios en Microservicios*

ID	Métrica	Express	Gin	Phoenix	Unidad / Nota

PCa-3-S	Adecuación al incremento del acceso de usuarios	0.547	0.577	0.575	Objetivo establecido: 750 VUs

Para finalizar con esta sección, se presenta la Figura 28 con los datos anteriores para un mejor entendimiento.

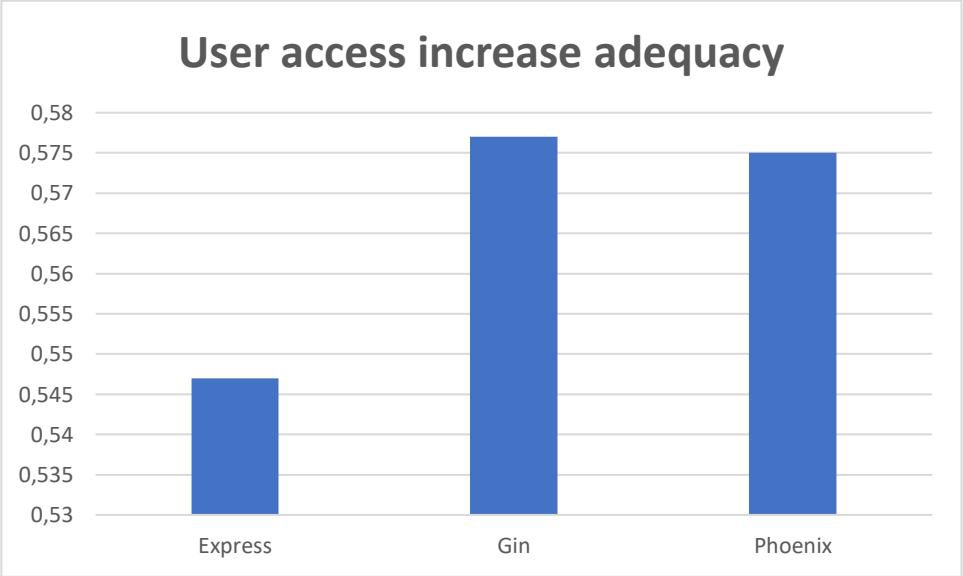


Figura 28 Adecuación al incremento del acceso de usuarios en Microservicios

4.3 Análisis comparativo global

A partir de los resultados consolidados en este capítulo, se evidencia que los frameworks evaluados manifiestan comportamientos divergentes y ventajas competitivas condicionadas, estrictamente por el tipo de operación y el estilo arquitectónico implementado.

4.3.1 Errores encontrados

Durante las pruebas de carga y estrés surgieron errores, los cuales probablemente fueron producidos por diversas causas, como la capacidad de las APIs, la base de datos y/o el fluctuante internet. Las pruebas de carga realizadas en las APIs de usuario tienen un porcentaje mínimo de errores, Express con 0,003%, Gin con 0,0006% y Phoenix con 0%, donde las causas más factibles son la saturación de la base de datos y el agotamiento del pool de conexiones. Del mismo modo, en las pruebas de carga en las APIs maestro-

detalle se evidencia las mismas causas, donde Phoenix tuvo un 13.22% de errores mientras que sus competidores presentaron un 0%.

Los errores en las pruebas de estrés son más notorios como hemos visto anteriormente, y las causas son las mismas, siendo la principal, la saturación de la base de datos, la cual alcanzaba picos máximos en uso de CPU al verificar el panel de render de la base de datos. Lo sorprendente aquí fue el desempeño de Gin, con la API de Usuarios, que tuvo un porcentaje cercano al 0% de errores, debido a una sola petición que fallo durante las 3 pruebas. Por otro lado, Phoenix tuvo 6,70% de errores debido a las causas anteriores y el timeout configurado; y Express con un 0,52% de errores. Las pruebas de estrés realizadas en las APIs maestro-detalle siguen una tendencia similar, donde Express tiene 0,63%, Gin un 0% y Phoenix un 11.25% de errores.

Por último, las pruebas de carga realizadas en los microservicios con el API Gateway, muestran algo interesante, debido a que no existen errores, pero hay una diferencia entre las peticiones totales y las peticiones exitosas. Dando a entender que el Gateway filtra peticiones o no las envía debido a la cantidad masiva de peticiones que se envían de K6. Además, el Gateway y la base de datos consumen pocos recursos. La misma situación sucede con las pruebas de estrés, donde Express cuenta con 0,035% de errores y sus competidores con 0%. A diferencias de lo anterior, se evidencia que el Gateway muestra picos máximos en el consumo de CPU, mientras que la base de datos muestra un consumo superior a los pocos recursos utilizados en la prueba de carga, dando a entender que el Gateway actúa como un cuello de botella.

4.3.2 Entorno Monolítico

En la arquitectura monolítica, el rendimiento de los entornos responde directamente a las capacidades nativas de sus respectivos lenguajes y motores de ejecución:

- **Comportamiento temporal y transferencia:** Los entornos Phoenix y Gin lideran las métricas de tiempo medio de respuesta y rendimiento medio, exhibiendo una ligera ventaja adaptativa de Phoenix en la velocidad de despacho de solicitudes.
- **Utilización de recursos:** En el uso de hardware, Gin manifiesta una superioridad notoria, consolidando el consumo de CPU y memoria RAM más bajo del experimento. En contraposición, Phoenix exhibe una penalización crítica en

memoria, requiriendo hasta diez veces más recursos que Gin, posiblemente debido al costo base de inicialización de la Máquina Virtual de Erlang (BEAM).

- **Tráfico de red:** El consumo de ancho de banda se mantiene paritario en escenarios simples. Sin embargo, ante la complejidad transaccional, Phoenix demuestra una gran superioridad en la recepción de datos, asimilando mayores tasas de transferencia sin degradar su estructura, mientras que Express reduce drásticamente su rendimiento de red.
- **Capacidad y resiliencia bajo estrés:** En operaciones simples, Phoenix tolera los mayores techos de concurrencia, aunque registrando una tasa de fallos incremental. No obstante, ante operaciones complejas, la tendencia se revierte: Express se posiciona como el entorno más resiliente al soportar el mayor volumen de usuarios con un margen de error mínimo (< 1%), mientras que Phoenix sufre un colapso estructural prematuro al superar el umbral límite del 10% de errores. En este escenario, Gin destaca por su efectividad operativa, logrando una tasa de errores aproximadamente del 0% en ambos escenarios.

4.3.3 Entorno de Microservicios

La transición hacia una arquitectura descentralizada transforma radicalmente las dinámicas de rendimiento, diluyendo las particularidades nativas de los frameworks bajo un efecto de homogeneización arquitectónica.

La introducción del API Gateway añade una latencia base que estandariza las métricas de comportamiento temporal, utilización de recursos y capacidad. Dentro de esta paridad, las diferencias son muy ligeras, destacando Phoenix en las métricas de comportamiento temporal, una similitud de los tres frameworks en las métricas de utilización de recursos y Gin destacando en las métricas de capacidad, pero existiendo una diferencia muy mínima que nos lleva pensar que el API Gateway es el cuello de botella que neutraliza las eficiencias nativas de estas tecnologías.

4.3.4 Síntesis Comparativa

El análisis cruzado permite concluir que la arquitectura monolítica maximiza el potencial y la identidad de cada framework, permitiendo que Gin despliegue su extrema ligereza de hardware, que Phoenix explote su velocidad de procesamiento en entornos simples y que Express demuestre su madurez en persistencias complejas.

Por el contrario, la arquitectura de microservicios traslada la responsabilidad del rendimiento desde el framework hacia la infraestructura. La capacidad de procesamiento, la resiliencia y el escalado de los microservicios internos quedan condicionados y limitados de forma directa por la capacidad de despacho del API Gateway, el cual se consolida como el principal cuello de botella técnico del ecosistema distribuido, independientemente de si el microservicio está construido sobre Express, Gin o Phoenix.

4.4 Discusión

La literatura científica revisada permite establecer puntos de convergencia y discrepancias tecnológicas con la investigación realizada, aportando una mayor comprensión del comportamiento de los entornos evaluados.

En el tema de utilización de recursos, específicamente el uso de memoria, Ardiansyah y Fatwanto [31] demostraron una ligera ventaja por parte de JavaScript al compararlo con Go, en dos condiciones distintas: una local y otra remota, empleando la solicitud GET, donde JavaScript obtuvo 18.93MB y 41.29MB; y Go obtuvo 21.50MB y 41.36MB en la condición remota. Este comportamiento difiere de los datos de la presente investigación, donde Express y Gin registraron 36.5MB y 16.6MB respectivamente, con una diferencia similar en el CRUD de Facturas (42.5MB y 24.8MB). Esta diferencia se fundamenta en la constante evolución de las versiones de Node.js y Go, debido a que el estudio citado fue publicado en 2022 y en el transcurso de 4 años, estas versiones recibieron actualizaciones y mejoras que se reflejan en este trabajo.

Por otro lado, Lion et al. [32] evaluaron distintos lenguajes como Java, Go, JavaScript, Python y C++, enfocándose en el entorno de ejecución, donde emplearon diferentes aplicaciones para medir los frameworks. Los resultados mostraron que C++ utilizando GCC fue superior en la mayoría de los escenarios, seguido por Go y Java, tanto en velocidad como en uso de memoria. JavaScript fue el siguiente por su desempeño, coincidiendo con el comportamiento de Express en esta investigación, debido a que obtuvo el peor rendimiento entre los tres frameworks, lo que abre un trabajo futuro para replicar esta metodología experimental utilizando frameworks basados en C++, evaluando qué tan cerca se posicionan del rendimiento de Gin y Phoenix.

Gallego [25] evaluó varios frameworks en diferentes aspectos, abarcando en uno el rendimiento, donde demuestra que JavaScript (Express) fue superior en la inserción y selección de imágenes en la aplicación web y servicio web, superando a PHP (Laravel),

Python (Flask) y Java (Spring Boot). En otra línea similar, Lara [27] comparó dos stacks tecnológicos: Spring Boot (Java) y MERN(JavaScript), en donde evaluó el rendimiento mediante pruebas de carga y estrés, evidenciando que MERN fue más eficiente. Por otro lado, Brar et al. [33] compararon a Python, PHP y Node.js, los cuales utilizaron tres escenarios para las pruebas, donde el que mayor ventaja tiene es Node.js seguido muy de cerca por PHP, y obteniendo los peores resultados Python. Concluyen que Node.js es adecuado para sitios web intensivos en E/S y manejo de concurrencia. No obstante, al comparar JavaScript con lenguajes destinados a manejar la concurrencia, los resultados son muy distintos como se puedo observar en la presente investigación, esta diferencia puede explicarse por los goroutines de Gin y los procesos ligeros de la máquina virtual de Erlang (BEAM) por parte de Phoenix, demostrando que ambos frameworks son superiores en el desarrollo de APIs de alta disponibilidad y escalabilidad.

Por su parte, Di Meglio y Starace [29] nos brindan otro punto de vista respecto a JavaScript. En su estudio evaluaron varios frameworks y entornos de ejecución distribuidos entre los lenguajes: Java, JavaScript/TypeScript y Python. Para medir el rendimiento, realizaron pruebas de carga y estrés, donde el mejor fue Express con los entornos de ejecución Bun y Node, ambos mostraron una superioridad frente a las otras 10 configuraciones en este aspecto, siendo Bun ligeramente mejor. Dicho estudio nos brinda un nuevo enfoque, debido a que, en mi investigación, Express con Node termino con los peores resultados en la mayoría de las métricas evaluadas. Dejando un camino, para trabajos futuros donde se evalúen frameworks dentro del mismo lenguaje, en este caso Go y Elixir, y así observar el mejor framework y su entono de ejecución.

Finalmente, el comportamiento de los endpoints evaluados ratificó patrones operativos estandarizados. Grudniak y Dzieńkowski [30] evaluaron dos frameworks basados en JavaScript: Express y Hapi, enfocándose en el rendimiento, contemplando las operaciones básicas (CRUD), los resultados mostraron una superioridad ligera por parte de Express en todos los escenarios a excepción de uno, que manejaba datos grandes, en donde quedaron con un rendimiento idéntico. Además, hacen énfasis que la operación PUT tomo más tiempo en completarse, a diferencia del GET que fue el de menor tiempo, y esto concuerda con la presente investigación en donde Express tiene la misma tendencia en esas operaciones.

CONCLUSIONES

El framework Gin se consolidó como el entorno más eficiente y confiable del estudio. Obtuvo una tasa de error aproximadamente del 0% tanto en escenarios monolíticos como en microservicios, operando con el menor consumo de CPU y memoria RAM. Su arquitectura basada en goroutines garantizó una alta disponibilidad, convirtiéndolo en la opción óptima para sistemas que exigen un estricto control de costes de hardware y tolerancia cero a fallos.

El ecosistema Phoenix exhibió una velocidad de procesamiento superior en escenarios de baja complejidad y alta concurrencia masiva. No obstante, esta velocidad se vio penalizada ante operaciones de alta carga transaccional en la base de datos, donde el entorno experimentó fallos estructurales prematuros y superó el umbral límite del 10% de errores en pruebas de estrés complejas debido a la saturación en la capa de persistencia.

A pesar de registrar los tiempos de respuesta más altos y la menor tasa de transferencia en escenarios simples, el framework Express demostró estabilidad y resiliencia adaptativa ante operaciones complejas (maestro-detalle). Soportó el mayor techo de usuarios concurrentes bajo estrés complejo (718 VUs) manteniendo una tasa de fallos menor al 1%, lo que evidenció la estabilidad de su ciclo de eventos (event loop) al gestionar colas de persistencia pesadas.

La arquitectura de microservicios demuestra que la incorporación de un API Gateway actúa como un factor de igualdad del rendimiento. Al centralizar el tráfico y añadir una latencia base de red, se mitigan las diferencias nativas de los frameworks, trasladando el cuello de botella técnico desde el código de la aplicación hacia la infraestructura de enrutamiento distribuido.

La complejidad de las transacciones altera notablemente el comportamiento de los frameworks. Mientras que las operaciones simples benefician el paralelismo de Phoenix, las operaciones complejas invierten la jerarquía de rendimiento, posicionando a Express y Gin como entornos más estables frente a la fatiga por almacenamiento de datos.

La aplicación sistemática de pruebas de carga y estrés, bajo el estándar de calidad ISO/IEC 25023, demuestra ser una metodología capaz de descubrir el comportamiento límite de los frameworks, permitiendo identificar que el rendimiento de un servicio no es absoluto, sino que depende estrictamente de la naturaleza de la transacción y el estilo arquitectónico.

RECOMENDACIONES

Para evitar cuellos de botella, se recomienda realizar un ajuste en los pools de conexiones de la base de datos. Es necesario ajustar el número de conexiones simultáneas permitidas por la base de datos con la capacidad de concurrencia nativa del framework, evitando así la saturación.

La capacidad de la base de datos juega un papel importante en la disponibilidad de un servicio. Por lo tanto, se recomienda distribuir adecuadamente los recursos, CPU y memoria RAM, en la base de datos para mitigar cuellos de botella. Debido a que, durante las pruebas de estrés el límite se alcanzó en la base de datos, provocando fallas en las transacciones y el aumento de la latencia.

Se sugiere mantener actualizadas las librerías utilizadas en los frameworks. Esto previene la introducción de latencias, garantizando un adecuado funcionamiento de las tecnologías, esto debido a que durante las pruebas se evidenció que Go, al emplear una librería diferente o actualizada, obtuvo mejoras considerables en los resultados.

Con el fin de profundizar en el rendimiento de software, se recomienda complementar esta investigación mediante la ejecución de Pruebas de Resistencia continuadas durante varias horas; para monitorizar la estabilidad de la memoria RAM y descartar fugas de recursos. De igual forma, se sugiere aplicar Pruebas de Pico para determinar la velocidad de recuperación y estabilización de los frameworks ante cargas masivas e instantáneas de tráfico.

Como trabajo futuro derivado de este estudio, se recomienda realizar un análisis comparativo que contraste el rendimiento de estos frameworks interactuando con motores de bases de datos NoSQL frente a los relacionales, evaluando cuál framework es el más eficiente bajo cargas normales y masivas.

REFERENCIAS BIBLIOGRÁFICAS

[1] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[2] M. G. López, "Integridad y control de concurrencia." Accessed: Oct. 09, 2025. [Online]. Available: https://sarreplec.caib.es/pluginfile.php/9867/mod_resource/content/4/BD05_Contenidos_Imprimible/IntegridadMariaGertrudis.pdf

[3] S. Salii, J. Ajdari, and X. Zenuni, "Migrating to a microservice architecture: benefits and challenges," in *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*, Opatija: IEEE, May 2023, pp. 1670–1677. doi: 10.23919/MIPRO57284.2023.10159894.

[4] W. P. Leones Zambrano, L. M. Macias Bazurto, W. I. Pilla Zuniga, and E. G. Fernández Sánchez, "Diseño Estratégico de APIs Escalables y Seguras para la Integración de Sistemas y Aplicaciones.," *Ciencia Latina Revista Científica Multidisciplinar*, vol. 8, no. 5, pp. 13395–13408, Dec. 2024, doi: 10.37811/cl_rcm.v8i5.14794.

[5] Naciones Unidas, "Objetivos de Desarrollo Sostenible." [Online]. Available: https://www.un.org/es/impacto-acad%C3%A9mico/page/objetivos-de-desarrollo-sostenible

[6] Secretaría Nacional de Planificación, "Plan de Creación de Oportunidades 2021-2025," 2021, Accessed: Oct. 22, 2025. [Online]. Available: https://www.planificacion.gob.ec/wp-content/uploads/2021/09/Plan-de-Creacio%CC%81n-de-Oportunidades-2021-2025-Aprobado.pdf

[7] V. M. López, S. A. Sánchez, and C. González, "Dificultad de empleo a jóvenes egresados sin experiencia," *Humanidades, tecnología y ciencia, del Instituto Politécnico Nacional*, 2021. [Online]. Available: https://revistaelectronica-ipn.org/ResourcesFiles/Contenido/26/HUMANIDADES_26_001009.pdf

[8] H. Subedi, "Managing Technical Debt In Your IT Infrastructure," Jones IT. [Online]. Available: https://www.itjones.com/blogs/2021/6/8/managing-technical-debt-in-your-it-infrastructure

[9] L. Alderman and S. Murray, “Benchmarking: Seeking Best Practice,” *New Dir. Eval.*, vol. 2025, no. 185–186, pp. 25–32, Mar. 2025, doi: 10.1002/ev.20634.

[10] V. Pavlova and I. Chebavova, “Implementation of digital technologies for managing the enterprise competitiveness based on benchmarking research,” *Economic Herald of SHEI USUCT*, vol. 21, no. 1, pp. 225–232, Jun. 2025, doi: 10.32434/2415-3974-2025-21-1-225-232.

[11] P. P. Patharlagadda, “Best Coding Practices to Improve Performance and Readability of Go Applications,” *Journal of Artificial Intelligence & Cloud Computing*, vol. 3, no. 1, pp. 1–6, Feb. 2024, doi: 10.47363/JAICC/2024(3)E113.

[12] D. Goldiner, “Application of GO programming language for simulation of mass service processes,” *INNOVATIVE TECHNOLOGIES AND SCIENTIFIC SOLUTIONS FOR INDUSTRIES*, no. 2(28), pp. 65–75, Jun. 2024, doi: 10.30837/2522-9818.2024.2.065.

[13] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, “The Go programming language and environment,” *Commun. ACM*, vol. 65, no. 5, pp. 70–78, May 2022, doi: 10.1145/3488716.

[14] M. Shoikhedbrod, “Using JavaScript in web design,” *BOHR International Journal of Engineering*, vol. 2, no. 1, pp. 7–14, 2023, doi: 10.54646/bije.2023.12.

[15] A. Shukla, “Modern JavaScript Frameworks and JavaScript’s Future as a FullStack Programming Language,” *Journal of Artificial Intelligence & Cloud Computing*, pp. 1–5, Dec. 2023, doi: 10.47363/JAICC/2023(2)144.

[16] S. Jurić, *Elixir in Action*, 1st ed. Shelter Island, NY: Manning Publications Co, 2015.

[17] J. Luo, B. Zhou, Y. Zheng, and W. Pan, “Research on high performance web service construction method based on JavaScript asynchronous programming technique,” *Applied Mathematics and Nonlinear Sciences*, vol. 9, no. 1, Jan. 2024, doi: 10.2478/amns-2024-2811.

[18] M. Niswar, R. A. Safruddin, A. Bustamin, and I. Aswad, “Performance evaluation of microservices communication with REST, GraphQL, and gRPC,” *International*

Journal of Electronics and Telecommunications, vol. 70, no. 2, pp. 429–436, Jun. 2024, doi: 10.24425/ijet.2024.149562.

[19] J. Jiang, "Algorithm Optimization and Performance Enhancement in Computer Science," *Frontiers in Computing and Intelligent Systems*, vol. 12, no. 2, pp. 70–72, 2025, doi: 10.54097/9t2gsv54.

[20] C. Yang, "Theoretical Analysis of Distributed Systems and Their Scalability," *Advances in Computer, Signals and Systems*, vol. 9, no. 1, 2025, doi: 10.23977/acss.2025.090104.

[21] N. Thallapally, "Improving System Availability and Load Balancing in Distributed Systems," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pp. 567–573, Jun. 2022, doi: 10.32628/cseit22547.

[22] Y. Wang, "Computation and Concurrency," pp. 1–87, Nov. 2025, doi: https://doi.org/10.48550/arXiv.2409.02595.

[23] A. Tornal Hidalgo, "Gestión de Infraestructura en la Nube: Herramientas, Complejidad y Despliegue," 2024. Accessed: Dec. 13, 2025. [Online]. Available: https://riunet.upv.es/server/api/core/bitstreams/a15bb7d6-90fd-4573-b0e0-1f35a9c31031/content

[24] L. Zavaleta Carranza, C. Verástegui Castillo, and A. Mendoza De Los Santos, "Beneficios y desafíos de la integración de infraestructura en la nube en la gestión de servicios de TI: una revisión sistemática", doi: 10.57166/riqchary/v7.n1.2025.

[25] V. Gallego Izquierdo, "Análisis de rendimiento de lenguajes de desarrollo de aplicaciones y servicios web," 2024. Accessed: Oct. 22, 2025. [Online]. Available: https://hdl.handle.net/2117/407069

[26] D. A. Yánez Flores and P. D. C. Jaramillo Flores, "Comparativa de Frameworks de Desarrollo Backend: estudio de caso entre Spring y Struts," *Revista Ingenio global*, vol. 4, no. 2, pp. 114–128, Sep. 2025, doi: 10.62943/rig.v4n2.2025.344.

[27] P. A. Lara Ortiz, "Comparativa de Stacks Java y Javascript para la implementación de un servicio web Restful para consulta de datos en la Universidad Católica Sede

Esmeraldas," 2022. [Online]. Available: https://repositorio.puce.edu.ec/items/a2547b88-bc79-41ef-a18c-88c9686c6877

[28] S. Hussein, S. J. Abbas, G. F. Ali, N. K. Hadi, and M. M. M. Maadi, "A Comparative Analysis of Programming Languages Used in Microservices," *International Journal of Computational and Experimental Science and Engineering*, vol. 11, no. 1, pp. 1203–1213, 2025, doi: 10.22399/ijcesen.977.

[29] S. Di Meglio and L. L. L. Starace, "Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies," *IEEE Access*, vol. 12, pp. 161649–161669, 2024, doi: 10.1109/ACCESS.2024.3489892.

[30] M. Grudniak and M. Dzieńkowski, "REST API performance comparison of web applications based on JavaScript programming frameworks," *Journal Computer Sciences Institute*, vol. 19, pp. 121–125, 2021, doi: https://doi.org/10.35784/jcsi.2620.

[31] H. Ardiansyah and A. Fatwanto, "Comparison of Memory usage between REST API in Javascript and Golang," *MATRIK : Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer*, vol. 22, no. 1, pp. 229–240, Nov. 2022, doi: 10.30812/matrik.v22i1.1325.

[32] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?," *2022 USENIX Annual Technical Conference*, pp. 835–852, 2022, [Online]. Available: https://www.usenix.org/conference/atc22/presentation/lion

[33] H. Brar, T. Kaur, and Y. Rajoria, "The Better Comparison between PHP, Python-web & Node.js," *International Journal of Research in Engineering and Science (IJRES)*, vol. 9, no. 7, pp. 29–37, 2021, Accessed: Jan. 20, 2026. [Online]. Available: https://api.semanticscholar.org/CorpusID:246002410

[34] ISO/IEC, *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality, ISO/IEC Standard 25023:2016*. 2016.

ANEXOS

En esta sección se presenta información extra que permite comprender de mejor manera la investigación.

Anexo 1: Script utilizado para las pruebas de carga en el CRUD de Usuarios (Monolítico).

```
        import http from 'k6/http';
        import { sleep, check } from 'k6';
        import { Trend, Rate, Counter } from 'k6/metrics';

        const responseTimePost   = new Trend('response_time_post',   true);
        const responseTimeGet    = new Trend('response_time_get',    true);
        const responseTimePut    = new Trend('response_time_put',    true);
        const responseTimeDelete = new Trend('response_time_delete', true);
        const failedRequests     = new Rate('failed_requests');
        const successfulOps      = new Counter('successful_operations');
        const errorRatePost      = new Rate('error_rate_post');
        const errorRateGet       = new Rate('error_rate_get');
        const errorRatePut       = new Rate('error_rate_put');
        const errorRateDelete    = new Rate('error_rate_delete');

        export const options = {
          scenarios: {
            prueba_carga: {
              executor:  'constant-vus',
              vus:       200,
              duration:  '4m',
              exec:      'testAPI',
            },
          },
          thresholds: {
            http_req_duration:    ['p(95)<2000', 'p(99)<3000'],
            response_time_post:   ['p(95)<1500'],
            response_time_get:    ['p(95)<800'],
            response_time_put:    ['p(95)<1500'],
            response_time_delete: ['p(95)<1000'],
            failed_requests:      ['rate<0.01'],
            successful_operations:['count>0'],
          },
        };

        export function testAPI() {
           const baseUrl = 'https://go-gin-6frs.onrender.com/users';
          // const baseUrl = 'https://js-express-unah.onrender.com/users';
          // const baseUrl = 'https://elixir-phoenix-9014.onrender.com/users';

          // const baseUrl = 'https://gatewaypy.onrender.com/api/v1/users';
          const params  = { headers: { 'Content-Type': 'application/json' } };

          // --- 1. OPERACIÓN POST ---
          const postRes = http.post(baseUrl,
            JSON.stringify({   email:   `user_${__VU}_${__ITER}@test.com`,   password:
'password123' }),
            params
          );
          responseTimePost.add(postRes.timings.duration);
          const postOk = check(postRes, { 'POST 200/201': (r) => r.status === 200 ||
r.status === 201 });

          errorRatePost.add(!postOk);
          failedRequests.add(!postOk);
```

```
        // Detener la iteración si el POST falló para evitar errores en cascada
        if (!postOk) {
          return;
        }
        successfulOps.add(1);

        // Extraer ID real de forma segura
        let userId;
        try {
          userId = String(JSON.parse(postRes.body).id);
        } catch (_) {
          failedRequests.add(true);
          return; // Si no hay ID válido, abortamos este ciclo
        }

        // --- 2. OPERACIÓN GET ---
        const getRes = http.get(http.url`${baseUrl}/${userId}`, params);
        responseTimeGet.add(getRes.timings.duration);
        const getOk = check(getRes, { 'GET 200': (r) => r.status === 200 });
        errorRateGet.add(!getOk);
        failedRequests.add(!getOk);
        if (getOk) successfulOps.add(1);
        sleep(0.5);

        // --- 3. OPERACIÓN PUT ---
        const putRes = http.put(http.url`${baseUrl}/${userId}`,
          JSON.stringify({   email:   `upd_${__VU}_${__ITER}@test.com`,   password:
'newpass123' }),
          params
        );
        responseTimePut.add(putRes.timings.duration);
        const putOk = check(putRes, { 'PUT 200': (r) => r.status === 200 });
        errorRatePut.add(!putOk);
        failedRequests.add(!putOk);
        if (putOk) successfulOps.add(1);
        sleep(0.5);

        // --- 4. OPERACIÓN DELETE ---
        const deleteRes = http.del(http.url`${baseUrl}/${userId}`, null, params);
        responseTimeDelete.add(deleteRes.timings.duration);
        const deleteOk = check(deleteRes, { 'DELETE 200/204': (r) => r.status ===
200 || r.status === 204 });
        errorRateDelete.add(!deleteOk);
        failedRequests.add(!deleteOk);
        if (deleteOk) successfulOps.add(1);
        sleep(1);
      }

      export function handleSummary(data) {
        const m = data.metrics;
        const v = (metricName, key) => {
          const metric = m[metricName];
          if (!metric || !metric.values) return null;
          return metric.values[key] ?? null;
        };
        const targets = { post: 1500, get: 800, put: 1500, delete: 1000 };

        const adequacy = (metric, target) => {
          const p95 = v(metric, 'p(95)');
          return p95 !== null ? (p95 / target).toFixed(4) : 'N/A';
        };

        const summary = {
          framework: 'EXPRESS', // cambia por GIN o PHOENIX según corresponda
```

```
            'PTb-1-G_Mean_Response_Time_ms': {
              POST_avg:   v('response_time_post',   'avg'),
              GET_avg:    v('response_time_get',    'avg'),
              PUT_avg:    v('response_time_put',    'avg'),
              DELETE_avg: v('response_time_delete', 'avg'),
              TOTAL_avg:  v('http_req_duration',    'avg'),
              POST_p95:   v('response_time_post',   'p(95)'),
              GET_p95:    v('response_time_get',    'p(95)'),
              PUT_p95:    v('response_time_put',    'p(95)'),
              DELETE_p95: v('response_time_delete', 'p(95)'),
            },

            'PTb-2-G_Response_Time_Adequacy': {
              POST:   adequacy('response_time_post',   targets.post),
              GET:    adequacy('response_time_get',    targets.get),
              PUT:    adequacy('response_time_put',    targets.put),
              DELETE: adequacy('response_time_delete', targets.delete),
              nota:   'X < 1 adecuado | X >= 1 no adecuado',
            },

            'PTb-5-G_Mean_Throughput': {
              total_requests: v('http_reqs', 'count'),
              rps:            v('http_reqs', 'rate'),
            },

            'PRu-4-S_Bandwidth_Utilization': {
              data_sent_bytes:       v('data_sent',     'count'),
              data_sent_rate_bps:    v('data_sent',     'rate'),
              data_received_bytes:   v('data_received', 'count'),
              data_received_rate_bps:v('data_received', 'rate'),
            },

            'PCa-1-G_Transaction_Processing_Capacity': {
              total_requests:   v('http_reqs',       'count'),
              failed_count:     v('failed_requests', 'passes'),
              error_rate:       v('failed_requests', 'rate'),
              nota: 'Revisar archivo .txt para ver en qué VU subió el error durante
ramp-up',
            },

            'PCa-2-G_User_Access_Capacity': {
              successful_operations: v('successful_operations', 'count'),
              max_vus_reached:       v('vus_max',               'max'),
            },

            'PCa-3-S_User_Access_Increase_Adequacy': {
              max_vus_reached: v('vus_max',  'max'),
              target_vus:      200,
              adequacy_ratio:  ((v('vus_max', 'max') || 0) / 200).toFixed(4),
              nota:            'X = VUs_alcanzados / 200',
            },

            'Error_Rates_By_Operation': {
              POST_error_rate:   v('error_rate_post',   'rate'),
              GET_error_rate:    v('error_rate_get',     'rate'),
              PUT_error_rate:    v('error_rate_put',     'rate'),
              DELETE_error_rate: v('error_rate_delete',  'rate'),
            },
          };

          const output = JSON.stringify(summary, null, 2);
          console.log('\n========== RESUMEN ISO 25023 ==========\n' + output);

          return {
            'stdout': output,
            'pcusuarioexpress1.json': JSON.stringify(
```

```
            { ...summary, raw_metrics: data.metrics }, null, 2
          ),
        };
      }
```

Anexo 2: Script utilizado para las pruebas de estrés en el CRUD de Usuarios (Monolítico).

```
      import http from 'k6/http';
      import { sleep, check } from 'k6';
      import { Trend, Rate, Counter } from 'k6/metrics';

      const responseTimePost = new Trend('response_time_post', true);
      const responseTimeGet = new Trend('response_time_get', true);
      const responseTimePut = new Trend('response_time_put', true);
      const responseTimeDelete = new Trend('response_time_delete', true);
      const failedRequests = new Rate('failed_requests');
      const successfulOps = new Counter('successful_operations');
      const errorRatePost = new Rate('error_rate_post');
      const errorRateGet = new Rate('error_rate_get');
      const errorRatePut = new Rate('error_rate_put');
      const errorRateDelete = new Rate('error_rate_delete');

      export const options = {
        scenarios: {
          breakpoint_test: {
            executor: 'ramping-vus',
            startVUs: 0,
            stages: [
              { duration: '60s', target: 200 },
              { duration: '60s', target: 400 },
              { duration: '60s', target: 600 },
              { duration: '60s', target: 800 },
              { duration: '60s', target: 1000 },
              { duration: '60s', target: 1200 },
              { duration: '60s', target: 1400 },
              { duration: '60s', target: 1600 },
              { duration: '60s', target: 1800 },
              { duration: '60s', target: 2000 },
              { duration: '60s', target: 2200 },
              { duration: '60s', target: 2400 },
              { duration: '60s', target: 2600 },
              { duration: '60s', target: 2800 },
              { duration: '60s', target: 3000 },
              { duration: '60s', target: 3000 }, // mantiene el pico para confirmar
colapso
              { duration: '10s', target: 0 }, // bajada
            ],
            exec: 'testAPI',
          },
        },

        // abortOnFail detiene el test cuando detecta colapso real
        thresholds: {
          failed_requests: [{
            threshold: 'rate<0.10',  // colapso = más del 10% de errores
            abortOnFail: true,
            delayAbortEval: '15s',
          }],
          http_req_duration:  [{  threshold:  'p(95)<5000',  abortOnFail:  true,
delayAbortEval: '15s' }],
          // Estas NO abortan, solo registran
```

```
            response_time_post: ['p(95)<3000'],
            response_time_get: ['p(95)<3000'],
            response_time_put: ['p(95)<3000'],
            response_time_delete: ['p(95)<3000'],
            successful_operations: ['count>0'],
          },
        };

        export function testAPI() {
           const baseUrl = 'https://go-gin-6frs.onrender.com/users';
          // const baseUrl = 'https://js-express-unah.onrender.com/users';
          // const baseUrl = 'https://elixir-phoenix-9014.onrender.com/users';

          // const baseUrl = 'https://gatewaypy.onrender.com/api/v1/users';
          const params = { headers: { 'Content-Type': 'application/json' } };

          // POST
          const postRes = http.post(baseUrl,
            JSON.stringify({   email:   `user_${__VU}_${__ITER}@test.com`,   password:
'password123' }),
            params
          );
          responseTimePost.add(postRes.timings.duration);
          const postOk = check(postRes, { 'POST 200/201': (r) => r.status === 200 ||
r.status === 201 });
          errorRatePost.add(!postOk);
          failedRequests.add(!postOk);
          // Detener la iteración si el POST falló para evitar errores en cascada
          if (!postOk) {
            return;
          }
          successfulOps.add(1);

          // Extraer ID real de forma segura
          let userId;
          try {
            userId = String(JSON.parse(postRes.body).id);
          } catch (_) {
            failedRequests.add(true);
            return; // Si no hay ID válido, abortamos este ciclo
          }

          // GET
          const getRes = http.get(http.url`${baseUrl}/${userId}`, params);
          responseTimeGet.add(getRes.timings.duration);
          const getOk = check(getRes, { 'GET 200': (r) => r.status === 200 });
          errorRateGet.add(!getOk);
          failedRequests.add(!getOk);
          if (getOk) successfulOps.add(1);
          sleep(0.5);

          // PUT
          const putRes = http.put(http.url`${baseUrl}/${userId}`,
            JSON.stringify({   email:   `upd_${__VU}_${__ITER}@test.com`,   password:
'newpass123' }),
            params
          );
          responseTimePut.add(putRes.timings.duration);
          const putOk = check(putRes, { 'PUT 200': (r) => r.status === 200 });
          errorRatePut.add(!putOk);
          failedRequests.add(!putOk);
          if (putOk) successfulOps.add(1);
          sleep(0.5);

          // DELETE
          const deleteRes = http.del(http.url`${baseUrl}/${userId}`, null, params);
```

```
          responseTimeDelete.add(deleteRes.timings.duration);
          const deleteOk = check(deleteRes, { 'DELETE 200/204': (r) => r.status ===
200 || r.status === 204 });
          errorRateDelete.add(!deleteOk);
          failedRequests.add(!deleteOk);
          if (deleteOk) successfulOps.add(1);
          sleep(1);
        }

        export function handleSummary(data) {
          const m = data.metrics;
          const v = (name, key) => m[name]?.values?.[key] ?? null;
          const adequacy = (metric, target) => {
            const p95 = v(metric, 'p(95)');
            return p95 !== null ? (p95 / target).toFixed(4) : 'N/A';
          };

          const summary = {
            framework: 'EXPRESS', // cambia por GIN o PHOENIX

            'PTb-1-G_Mean_Response_Time_ms': {
              POST_avg: v('response_time_post', 'avg'),
              GET_avg: v('response_time_get', 'avg'),
              PUT_avg: v('response_time_put', 'avg'),
              DELETE_avg: v('response_time_delete', 'avg'),
              TOTAL_avg: v('http_req_duration', 'avg'),
              POST_p95: v('response_time_post', 'p(95)'),
              GET_p95: v('response_time_get', 'p(95)'),
              PUT_p95: v('response_time_put', 'p(95)'),
              DELETE_p95: v('response_time_delete', 'p(95)'),
            },

            'PTb-2-G_Response_Time_Adequacy': {
              POST: adequacy('response_time_post', 3000),
              GET: adequacy('response_time_get', 3000),
              PUT: adequacy('response_time_put', 3000),
              DELETE: adequacy('response_time_delete', 3000),
              nota: 'X < 1 adecuado | X >= 1 no adecuado',
            },

            'PTb-5-G_Mean_Throughput': {
              total_requests: v('http_reqs', 'count'),
              rps: v('http_reqs', 'rate'),
            },

            'PRu-4-S_Bandwidth_Utilization': {
              data_sent_bytes: v('data_sent', 'count'),
              data_sent_rate_bps: v('data_sent', 'rate'),
              data_received_bytes: v('data_received', 'count'),
              data_received_rate_bps: v('data_received', 'rate'),
            },

            'PCa-1-G_Transaction_Processing_Capacity': {
              total_requests: v('http_reqs', 'count'),
              failed_count: v('failed_requests', 'passes'),
              error_rate: v('failed_requests', 'rate'),
              nota: 'El test se detuvo automáticamente al superar 10% de errores o
p95 > 5000ms',
            },

            'PCa-2-G_User_Access_Capacity': {
              successful_operations: v('successful_operations', 'count'),
              max_vus_reached: v('vus_max', 'max'),
              nota: 'max_vus_reached = VUs activos cuando abortOnFail detuvo el test
= punto de quiebre',
            },
```

```
    'PCa-3-S_User_Access_Increase_Adequacy': {
      max_vus_reached: v('vus_max', 'max'),
      target_vus: 3000,
      adequacy_ratio: ((v('vus_max', 'max') || 0) / 3000).toFixed(4),
      incremento: '200 VUs cada 60 segundos',
    },

    'Error_Rates_By_Operation': {
      POST_error_rate: v('error_rate_post', 'rate'),
      GET_error_rate: v('error_rate_get', 'rate'),
      PUT_error_rate: v('error_rate_put', 'rate'),
      DELETE_error_rate: v('error_rate_delete', 'rate'),
    },
  };

  const output = JSON.stringify(summary, null, 2);
  console.log('\n========== BREAKPOINT TEST ISO 25023 ==========\n' + output);

  return {
    'stdout': output,
    'peusuariophoenix3.json': JSON.stringify(
      { ...summary, raw_metrics: data.metrics }, null, 2
    ),
  };
}
```

Anexo 3: Formato del Archivo JSON generado por K6 una vez terminado cada prueba, donde se puede observar una fracción de los resultados debido a que el archivo contiene aproximadamente 370 líneas de código.


```
{
  "framework": "EXPRESS",
  "PTb-1-G_Mean_Response_Time_ms": {
    "POST_avg": 238.64882977060506,
    "GET_avg": 235.26340383668688,
    "PUT_avg": 243.74528096134244,
    "DELETE_avg": 231.86359853771472,
    "TOTAL_avg": 237.3803930547794,
    "POST_p95": 457.2894,
    "GET_p95": 431.5261349999994,
    "PUT_p95": 456.7381599999997,
    "DELETE_p95": 407.3090349999999
  },
  "PTb-2-G_Response_Time_Adequacy": {
    "POST": "0.3049",
    "GET": "0.5394",
    "PUT": "0.3045",
    "DELETE": "0.4073",
    "nota": "X < 1 adecuado | X >= 1 no adecuado"
  },
  "PTb-5-G_Mean_Throughput": {
    "total_requests": 55261,
    "rps": 227.15035300996496
  },
  "PRu-4-S_Bandwidth_Utilization": {
    "data_sent_bytes": 5006925,
    "data_sent_rate_bps": 20580.96634596585,
    "data_received_bytes": 13976274,
    "data_received_rate_bps": 57449.477440943796
  },
```

```
          "PCa-1-G_Transaction_Processing_Capacity": {
            "total_requests": 55261,
            "failed_count": 5,
            "error_rate": 0.000090479723493965,
            "nota": "Revisar resultado_txt para ver en qué VU subió el error durante
ramp-up"
          },
          "PCa-2-G_User_Access_Capacity": {
            "successful_operations": 55256,
            "max_vus_reached": 200
          },
          "PCa-3-S_User_Access_Increase_Adequacy": {
            "max_vus_reached": 200,
            "target_vus": 200,
            "adequacy_ratio": "1.0000",
            "nota": "X = VUs_alcanzados / 200"
          },
          "Error_Rates_By_Operation": {
            "POST_error_rate": 0.0003618206816701643,
            "GET_error_rate": 0,
            "PUT_error_rate": 0,
            "DELETE_error_rate": 0
  },…
```


Anexo 4: Script de creación de tablas en la base de datos.

```
        CREATE TABLE users (
            id SERIAL PRIMARY KEY,
            email VARCHAR(100) ,
            password VARCHAR(100))

        -- 1. Crear la tabla Maestro (Facturas)
        CREATE TABLE facturas (
            id SERIAL PRIMARY KEY,
            num_factura VARCHAR(50) NOT NULL,
            customer VARCHAR(255) NOT NULL,
            employee VARCHAR(255) NOT NULL
        );

        -- 2. Crear la tabla Detalle (Detalles)
        CREATE TABLE detalles (
            id SERIAL PRIMARY KEY,
            factura_id INTEGER NOT NULL,
            product VARCHAR(255) NOT NULL,
            quantity INTEGER NOT NULL,
            price FLOAT NOT NULL,
            total FLOAT NOT NULL,

            CONSTRAINT fk_factura
                FOREIGN KEY(factura_id)
                REFERENCES facturas(id)
                ON DELETE CASCADE
        );
```

Anexo 5: Repositorios públicos, con el código fuente, los cuales fueron sometidos a las pruebas de carga y estrés.

Monolíticos
Gin:

https://github.com/sepazminot/Go-Gin-.git

Express:

https://github.com/sepazminot/JS-Express-.git

Phoenix:

https://github.com/sepazminot/elixir_phoenix.git

Microservicios

Gin:

https://github.com/sepazminot/MicroservicioGo.git

Express:

https://github.com/sepazminot/user-service.git

Phoenix:

https://github.com/sepazminot/MicroservicioElixir.git

API Gateway:

https://github.com/sepazminot/GatewayPy.git

Anexo 6: Captura de pantalla del panel de Render, luego de realizar 3 pruebas de carga, para registrar los picos más altos en el consumo de recursos.



Figura 29 Panel de Render