Descripción del Proyecto

El objetivo principal del proyecto es crear una aplicación web que facilite a los pacientes la programación de citas médicas. La plataforma debe ser intuitiva, segura y escalable, permitiendo la gestión eficiente de información tanto para pacientes como para doctores.

Tecnologías Utilizadas

Backend
Tecnologías Core:

- Node.js + TypeScript: Lenguaje y entorno de ejecución para desarrollar el servidor.
- Express.js: Framework para construir la API REST.
- Prisma: ORM para gestionar la interacción con la base de datos.

Arquitectura:

- Patrón Repository: Para separar la lógica de acceso a datos de la lógica de negocio.
- Controladores REST: Manejo de las solicitudes HTTP.
- Sistema de Rutas Modular: Organización de las rutas de la API de manera escalable.
- Middleware Personalizado: RateLimiter y PerfromanceMiddleware.
- Manejo de Errores Centralizado: Gestión uniforme de errores en toda la aplicación.

Frontend
Framework y Core:

- Next.js 14 con App Router: Framework React para aplicaciones web con renderizado del lado del servidor.
- React 18: Biblioteca para construir interfaces de usuario.
- TypeScript: Superset de JavaScript que añade tipado estático.
- Tailwind CSS: Framework de CSS utilitario para estilos rápidos y responsivos.

UI y Componentes:

- shadcn/ui: Biblioteca de componentes UI personalizados.
- Sistema de Componentes Modular: Componentes reutilizables y mantenibles.
- Layouts Reutilizables: Estructuras de página consistentes.
- Axios: Cliente HTTP para realizar peticiones.

Estructura de Carpetas:

- /app - Rutas y páginas.
- /components - Componentes reutilizables.
- /hooks - Custom hooks.
- /lib - Utilidades.

Bases de Datos y Docker

Contenedores:

- MongoDB Container: Base de datos NoSQL.
- PostgreSQL Container: Base de datos relacional.
- Volúmenes Persistentes: Almacenamiento de datos persistente.
- Redes Docker Aisladas: Comunicación segura entre contenedores.

MongoDB:

- Colecciones: Patients, Doctors, Appointments.
- Índices Optimizados: Mejora del rendimiento de consultas.
- Esquema Flexible: Adaptabilidad de la estructura de datos.
- Relaciones mediante Referencias: Enlaces entre diferentes colecciones.

Metodología de Desarrollo

El desarrollo del proyecto se estructuró en sprints, siguiendo una metodología ágil que permitió adaptarse a los requerimientos y mejorar continuamente la arquitectura y funcionalidades de la aplicación.

Estructura de Sprints

Cada sprint tuvo una duración de 2-3 semanas y se enfocó en objetivos específicos, garantizando entregas funcionales y mejoras incrementales en cada iteración.

Detalle de Sprints

Sprint 1: Configuración Inicial del Backend

Objetivos:

- Configurar el entorno de desarrollo.
- Establecer la estructura básica del backend.
- Configurar Docker para PostgreSQL.

Actividades Realizadas:

Configuración del Entorno:

- Instalación de Node.js y TypeScript.
- Inicialización del proyecto con npm init.
- Configuración de TypeScript (tsconfig.json).
- Configuración de Docker:
- Creación de contenedores para PostgreSQL.
- Configuración de volúmenes persistentes para la base de datos.
- Establecimiento de redes Docker aisladas para comunicación segura.

Configuración de Express.js:

- Instalación y configuración de Express.js.
- Configuración de scripts de inicio y desarrollo.

Integración de Prisma:

- Instalación de Prisma y configuración inicial.
- Definición del esquema de datos para PostgreSQL.
- Generación de migraciones iniciales.

Resultados:

- Backend configurado y listo para el desarrollo de funcionalidades.
- PostgreSQL corriendo en un contenedor Docker con acceso configurado.

Sprint 2: Implementación del CRUD con PostgreSQL

Objetivos:

- Implementar operaciones CRUD para las entidades principales (Patients, Doctors, Appointments).
- Establecer rutas y controladores básicos.

Actividades Realizadas:
Modelo de Datos:

- Definición de modelos Prisma para Patients, Doctors y Appointments.
- Migración de la base de datos.

Patrón Repository:

- Implementación de repositorios para cada entidad.
- Métodos básicos de acceso a datos (crear, leer, actualizar, eliminar).

Controladores REST:

- Desarrollo de controladores para manejar solicitudes CRUD.
- Validación de datos de entrada.

Rutas Modulares:

- Configuración de rutas para cada entidad.
- Uso de middleware para manejo de autenticación y logging.

Frontend Inicial:

- Configuración básica de Next.js con páginas para listar y gestionar entidades.
- Implementación de formularios básicos para CRUD.

Resultados:

- Sistema completamente funcional con operaciones CRUD implementadas para PostgreSQL.
- Sin embargo, se utilizó el formato de Next.js que comparte el backend y frontend en la misma aplicación sin una diferenciación clara entre ambos, lo que complicó la escalabilidad y mantenibilidad.

Reflexión:Este sprint mostró la importancia de planificar la arquitectura desde el inicio. La falta de separación entre frontend y backend resultó en una base de código menos modular, lo que dificultó la implementación de mejoras posteriores.

Sprint 3: Integración del Frontend con Next.js
Objetivos:

- Mejorar la interacción entre el frontend y el backend.
- Optimizar la gestión de estado y las peticiones HTTP.

Actividades Realizadas:
Configuración de Axios:

- Instalación y configuración de Axios para manejar peticiones HTTP al backend.

Componentes Reutilizables:

- Desarrollo de componentes modulares con shadcn/ui.
- Creación de layouts reutilizables para consistencia en la interfaz.

Optimización de Páginas:

- Mejoras en la experiencia de usuario mediante la implementación de revalidaciones y actualizaciones en tiempo real.
- Estilización con Tailwind CSS:
- Refinamiento de estilos para una interfaz más atractiva y responsiva.

Resultados:

- Frontend más robusto y eficiente en la gestión de datos.
- Mejoras significativas en la experiencia de usuario gracias a la optimización de componentes y estilos.

Reflexión: Axios mejoró considerablemente la eficiencia de las peticiones y la gestión del estado, facilitando una interacción más fluida entre frontend y backend.

Sprint 4: Optimización de la Arquitectura y Pruebas de SQL
Objetivos:

- Reestructurar el proyecto para cumplir con la arquitectura por capas de repositorios.
- Optimizar las consultas SQL realizando pruebas con y sin índices.

Actividades Realizadas:

Reestructuración de la Arquitectura:

- Implementación del Patrón Repository para separar la lógica de acceso a datos de la lógica de negocio.
- Refactorización de controladores para interactuar con los repositorios.
- Modularización adicional del sistema de rutas para mejorar la mantenibilidad.

Optimización de Consultas SQL:
- Análisis de consultas actuales para identificar posibles cuellos de botella.
- Implementación de índices en PostgreSQL y realización de pruebas de rendimiento.
- Comparación de tiempos de respuesta con y sin índices.

Resultados:
- Arquitectura más robusta y mantenible gracias a la separación de responsabilidades.
- Mejora en el rendimiento de las consultas SQL mediante la optimización de índices, reduciendo tiempos de respuesta y aumentando la eficiencia de la base de datos.

Reflexión: La reestructuración de la arquitectura permitió una mayor modularidad y facilidad de mantenimiento. Las pruebas de optimización de consultas SQL demostraron una mejora significativa en el rendimiento, validando la importancia de una base de datos bien estructurada.

Sprint 5: Migración a MongoDB
Objetivos:
- Migrar la base de datos de PostgreSQL a MongoDB.
- Adaptar la arquitectura existente para soportar la nueva base de datos.

Actividades Realizadas:
Planificación de la Migración:
- Análisis de las diferencias entre PostgreSQL y MongoDB.
- Identificación de las modificaciones necesarias en los modelos de datos.

Implementación con Prisma:
- Configuración de Prisma para trabajar con MongoDB.
- Adaptación de los repositorios para manejar el esquema flexible de MongoDB.

Migración de Datos:
- Desarrollo de scripts para migrar datos de PostgreSQL a MongoDB.
- Verificación de la integridad y consistencia de los datos migrados.

Refactorización Mínima:
- Reutilización de la estructura de repositorios creada en el Sprint 4.
- Realización de cambios mínimos en el código gracias a la arquitectura modular.

Resultados:
- Migración exitosa a MongoDB, mejorando la flexibilidad y escalabilidad de la base de datos.

- Proceso de migración simplificado gracias a Prisma y la arquitectura previamente establecida, permitiendo la reutilización de componentes con mínimos cambios.

Reflexión: La planificación cuidadosa y la arquitectura modular implementada en sprints anteriores facilitaron enormemente la migración a MongoDB. Prisma demostró ser una herramienta esencial para manejar diferentes tipos de bases de datos sin incurrir en grandes refactorizaciones.

Sprint 6: Pruebas de Rendimiento y Profiling
Objetivos:
- Realizar pruebas de perfilado y carga para asegurar la estabilidad y el rendimiento de la aplicación.
- Implementar mejoras basadas en los resultados de las pruebas.

Actividades Realizadas:
Implementación de Herramientas de Profiling:
- Configuración de Profiling Clinic para identificar cuellos de botella en el código.
- Integración de Autocannon para realizar pruebas de carga y evaluar la capacidad de la aplicación bajo diferentes niveles de tráfico.

Ejecución de Pruebas de Carga:
- Realización de pruebas de carga con distintos escenarios de tráfico.

Monitoreo de respuestas del sistema y recopilación de métricas de rendimiento.
- Análisis y Optimización:
- Análisis de los resultados obtenidos de las pruebas de profiling y carga.
- Implementación de optimizaciones en el código y la configuración del servidor para mejorar el rendimiento.

Documentación de Resultados:
- Generación de informes detallados sobre los resultados de las pruebas.
- Registro de las optimizaciones realizadas y sus impactos en el rendimiento.

Resultados:
- Obtención de buenos resultados en las pruebas de carga y profiling, asegurando que la aplicación puede manejar un alto volumen de solicitudes sin degradar su funcionamiento.
- Identificación y resolución de cuellos de botella, mejorando la eficiencia general de la aplicación.
- La migración a MongoDB se ejecutó de manera sencilla gracias a Prisma y la estructura modular, permitiendo reutilizar componentes con mínimos cambios.

Reflexión: Las pruebas de rendimiento y profiling fueron cruciales para garantizar la estabilidad y eficiencia de la aplicación en entornos de producción. Los buenos resultados obtenidos

validan la efectividad de las optimizaciones implementadas y la robustez de la arquitectura del proyecto.

Reflexiones y Retos

Durante el desarrollo del proyecto, enfrentamos varios desafíos que fueron superados gracias a una planificación cuidadosa y una metodología ágil:

Separación de Frontend y Backend:

- Reto: Inicialmente, utilizamos el formato de Next.js que compartía el backend y frontend en la misma aplicación, lo que complicó la escalabilidad.
- Solución: En sprints posteriores, reestructuramos la arquitectura para separar claramente las responsabilidades del frontend y backend, adoptando una arquitectura por capas de repositorios que facilitó futuras migraciones y mejoras.

Migración de Base de Datos:

- Reto: Migrar de PostgreSQL a MongoDB podría haber sido complejo debido a las diferencias entre bases de datos relacionales y NoSQL.
- Solución: Gracias a Prisma y a la organización modular implementada en el Sprint 4, la migración fue sencilla, permitiendo reutilizar gran parte del código con mínimos cambios.

Optimización de Rendimiento:

- Reto: Garantizar que la aplicación soportara un alto volumen de tráfico sin degradar su rendimiento.
- Solución: Implementamos herramientas de profiling y pruebas de carga que nos permitieron identificar y solucionar cuellos de botella, obteniendo buenos resultados en las pruebas de rendimiento.

Gestión de Estado y Peticiones HTTP:

- Reto: Manejar eficientemente el estado de la aplicación y las peticiones al backend.
- Solución: Axios mejoró significativamente la gestión de datos y el rendimiento de las peticiones HTTP.

Conclusiones

El desarrollo de la web app para agendar citas con doctores se realizó de manera estructurada y eficiente mediante la utilización de tecnologías modernas y una metodología ágil basada en sprints. La adopción del Patrón Repository y el uso de Prisma como ORM permitieron una arquitectura limpia y escalable, facilitando futuras expansiones y mantenimientos.

La migración de PostgreSQL a MongoDB se ejecutó sin contratiempos gracias a la planificación y la arquitectura previamente establecida, demostrando la importancia de una buena estructura de código desde las etapas iniciales del proyecto. Además, la implementación de pruebas de rendimiento aseguró que la aplicación cumple con los estándares necesarios para operar en un entorno de producción robusto.

En resumen, el proyecto alcanzó sus objetivos iniciales, proporcionando una solución efectiva para la gestión de citas médicas, y estableció una base sólida para futuras mejoras y escalabilidad.

Testing

SQL sin índices

Profiling:
CPU Usage:

- Picos de uso de CPU alcanzan el 600% al inicio, reflejando el alto costo de procesamiento debido a escaneos completos de tablas.
- La CPU se estabiliza en valores bajos después de procesar la carga inicial.

Memory Usage:

- Heap Used fluctúa entre 0 MB y 300 MB durante las operaciones iniciales, indicando un consumo considerable de memoria para manejar las consultas.

Event Loop Delay:

- Retraso máximo: 800 ms al inicio.
- Promedio en la operación: <200 ms, pero solo después de superar los cuellos de botella iniciales.

Active Handles:

- Pico inicial: 100 handles activos.
- Reducción rápida a casi 0 handles tras completar las operaciones principales, lo que podría deberse a desconexiones provocadas por timeouts.

Event Loop Utilization:

- Comienza en 100%, lo que indica saturación total, y decrece progresivamente hasta estabilizarse cerca del 10%, reflejando una carga inicial intensa.

Load Testing:

GET /api/appointments:

- Latencia promedio: 907 ms, con picos máximos de 9976 ms.
- Errores: 730 timeouts sobre un total de 26k solicitudes procesadas en 30 segundos.

GET /api/doctors/available-slots:

- Latencia promedio: 470 ms, con picos de hasta 6742 ms.
- Solicitudes procesadas: 64k en 30 segundos, con 0 errores, mostrando mejor manejo que otros endpoints.

GET /api/patients:

- Latencia promedio: 588 ms, con picos máximos de 9974 ms.
- Errores: 900 timeouts sobre 37k solicitudes procesadas.

GET /api/doctors:

- Latencia promedio: 516 ms, con picos máximos de 9982 ms.
- Errores: 60 timeouts sobre 57k solicitudes.

GET /api/health:

- Latencia promedio: 152 ms, con un máximo de 3486 ms.
- Solicitudes procesadas: 197k en 30 segundos, con 0 errores.

POST /api/appointments:

- Latencia promedio: 0.86 ms, con un máximo de 45 ms.
- Solicitudes procesadas: 3874 por segundo, sin errores ni timeouts.

Insights clave:

Latencias altas en consultas GET intensivas:

- Consultas como /api/appointments y /api/patients presentan latencias promedio superiores a 500 ms, con picos máximos cerca de los 10 segundos, lo cual es inaceptable para un sistema en producción.

Errores por timeout:

- 730 timeouts en /api/appointments y 900 timeouts en /api/patients, lo que representa el 2.8% y 2.4% del total de solicitudes, respectivamente, reflejando una ineficiencia crítica.

Mejor rendimiento en consultas simples:

- Endpoints como /api/health tienen latencias promedio de solo 152 ms, lo que demuestra que las consultas ligeras no sufren tanto por la falta de índices.

Diferencia en volumen procesado:

- Endpoints como /api/doctors/available-slots manejan 64k solicitudes en 30s, mientras que /api/appointments solo alcanza 26k solicitudes debido a la mayor complejidad de sus consultas.

Operaciones de escritura eficientes:

- El endpoint POST /api/appointments muestra un rendimiento excelente con una latencia promedio de 0.86 ms y 0 errores. Esto confirma que la falta de índices afecta principalmente a las operaciones de lectura complejas.

SQL con índices:

Profiling:

CPU Usage:

- Los picos iniciales de uso de CPU se reducen significativamente en comparación con el escenario sin índices. Aunque aún hay picos de 500%, estos son menos frecuentes y se estabilizan rápidamente.
- El uso general de CPU es más consistente, lo que indica que las consultas son más eficientes.

Memory Usage:

- Heap Used muestra fluctuaciones menores, con un consumo máximo de 300 MB, similar al escenario sin índices. Sin embargo, las operaciones son más estables y los picos son menos pronunciados.
- La asignación de memoria (RSS y Heap Allocated) se mantiene constante tras los picos iniciales, indicando un mejor manejo de recursos.

Event Loop Delay:

- No se observan retrasos significativos en el event loop (<10ms), en contraste con los 800ms del escenario sin índices. Esto confirma que el sistema responde más rápido a las solicitudes.

Active Handles:

- El número de handles activos también disminuye rápidamente después de los picos iniciales, pero se mantiene más estable, mostrando un mejor manejo de la concurrencia.

Event Loop Utilization:

- La utilización del event loop comienza alta (~80-90%) pero decrece más rápido y se estabiliza cerca del 10%, reflejando un procesamiento más eficiente.

Load Testing:

GET /api/appointments:

- Latencia promedio: 85ms, reducida drásticamente de los 907ms en el escenario sin índices.
- Errores: 0 timeouts, en comparación con los 730 errores anteriores.
- Solicitudes procesadas: 100k solicitudes en 30 segundos, frente a las 26k solicitudes sin índices.

GET /api/doctors/available-slots:

- Latencia promedio: 47ms, frente a los 470ms del caso sin índices.
- Solicitudes procesadas: 120k en 30 segundos, aumentando de 64k en el caso anterior.

GET /api/patients:

- Latencia promedio: 52ms, en comparación con los 588ms sin índices.
- Solicitudes procesadas: 98k en 30 segundos, en lugar de 37k.
- Errores: 0 timeouts, eliminando los 900 errores previos.

GET /api/doctors:

- Latencia promedio: 51ms, frente a 516ms sin índices.
- Solicitudes procesadas: 110k en 30 segundos, en contraste con las 57k previas.

GET /api/health:

- Latencia promedio: 10ms, en lugar de los 152ms del caso sin índices.
- Solicitudes procesadas: 200k en 30 segundos, una mejora marginal sobre las 197k sin índices.

POST /api/appointments:

- El rendimiento en operaciones de escritura no mostró cambios significativos, con latencias promedio de 0.86ms en ambos casos.

Insights clave:

Impacto de los índices:

- Los índices mejoraron drásticamente la latencia de las consultas GET, con reducciones de latencia de entre 90-95% y eliminando completamente los errores por timeout.

Aumento en solicitudes procesadas:

- Con índices, la cantidad de solicitudes procesadas aumentó entre 165% y 284%, dependiendo del endpoint.

Estabilidad del sistema:

- La estabilidad del sistema mejoró significativamente, con menor uso de CPU y eliminación de bloqueos en el event loop.

POST sin cambios:

- Las operaciones de escritura no se beneficiaron directamente de los índices, lo cual es esperado dado que no afectan directamente las inserciones.

NoSQL

Profiling

CPU Usage:

- Los picos iniciales son altos, alcanzando aproximadamente 400%-500%, pero con menor duración y frecuencia que en SQL sin índices. Esto muestra que NoSQL maneja mejor las consultas, aunque sigue exigiendo recursos intensivamente.
- Uso más consistente durante la carga, indicando una mayor estabilidad.

Memory Usage:

- La memoria utilizada (Heap Used) tiene fluctuaciones moderadas con un máximo de 400 MB, similar a SQL con índices, aunque el uso parece estabilizarse más rápidamente.
- RSS y Total Heap Allocated también muestran mayor estabilidad que en SQL sin índices.

Event Loop Delay:

- Los retrasos en el Event Loop son mínimos (<800ms).

Active Handles:

- Se observan picos de hasta 100 handles activos, similares a SQL, pero los handles se estabilizan rápidamente, mostrando un buen manejo de concurrencia.

Event Loop Utilization:

- La utilización del Event Loop comienza alta (~90%) pero desciende de manera uniforme y se estabiliza por debajo del 20%, reflejando eficiencia tras la carga inicial.

Load Testing:
GET /api/appointments:

- Latencia promedio: 1086ms, más alta que SQL con índices (85ms) pero similar a SQL sin índices (907ms).
- Solicitudes procesadas: 28k en 30 segundos, cerca de SQL sin índices (26k) pero menor que SQL con índices (100k).

GET /api/doctors/available-slots:

- Latencia promedio: 518ms, mejor que SQL sin índices (470ms) y acercándose a SQL con índices (47ms).
- Solicitudes procesadas: 58k en 30 segundos, igual que SQL sin índices y menor que SQL con índices (120k).

GET /api/patients:

- Latencia promedio: 820ms, más alta que SQL con índices (52ms) pero similar a SQL sin índices (588ms).
- Solicitudes procesadas: 37k en 30 segundos, igual que SQL sin índices y menor que SQL con índices (98k).

GET /api/doctors:

- Latencia promedio: 523ms, cercana a SQL sin índices (516ms) y más alta que SQL con índices (51ms).
- Solicitudes procesadas: 58k en 30 segundos, igual que SQL sin índices y menor que SQL con índices (110k).

POST /api/appointments:

- Latencia promedio: 1.44ms, ligeramente superior a SQL (0.86ms).
- Solicitudes procesadas: 2500 por segundo, menor que SQL (3873 por segundo).

Insights clave
Latencias moderadas:

- Las latencias promedio son menores que en SQL sin índices, pero más altas que en SQL con índices. Ejemplo: /api/appointments tiene una latencia promedio de 1086ms frente a 85ms en SQL con índices.

Volumen intermedio de solicitudes:

- Aunque mejora frente a SQL sin índices, NoSQL procesa menos solicitudes que SQL con índices. Ejemplo: /api/doctors/available-slots maneja 58k solicitudes frente a 120k en SQL con índices.

Estabilidad aceptable:

- Los picos de Event Loop Delay son bajos (<800ms), y el sistema muestra estabilidad, con manejo eficiente de concurrencia.

Sin errores ni timeouts:

- Al igual que SQL con índices, NoSQL maneja las solicitudes sin errores, mostrando un diseño robusto para la carga concurrente.

Análisis de los 3 escenarios

Métrica	SQL Sin Índices	SQL Con Índices	NoSQL
CPU Usage (pico)	600%	500%	400%-500%
Heap Used (MB)	300 MB	300 MB	400 MB
Event Loop Delay	800ms	<10ms	<800ms
Active Handles	Pico 100, inestable	Pico 100, estable	Pico 100, estable
GET /appointments	907ms, 730 timeouts, 26k	85ms, 0 timeouts, 100k	1086ms, 0 timeouts, 28k
GET /available-slots	470ms, 0 timeouts, 64k	47ms, 0 timeouts, 120k	518ms, 0 timeouts, 58k
GET /patients	588ms, 900 timeouts, 37k	52ms, 0 timeouts, 98k	820ms, 0 timeouts, 37k
GET /doctors	516ms, 60 timeouts, 57k	51ms, 0 timeouts, 110k	523ms, 0 timeouts, 58k
GET /health	152ms, 0 errores, 197k	10ms, 0 errores, 200k	N/A
POST /appointments	0.86ms, 0 errores, 3873/s	0.86ms, 0 errores, 3873/s	1.44ms, 0 errores, 2500/s

Impacto de los índices en SQL:

- La inclusión de índices en SQL transforma radicalmente el rendimiento, reduciendo las latencias promedio hasta en un 90% y aumentando el volumen de solicitudes procesadas en más de 300% en algunos endpoints.
- La estabilidad del sistema también mejora significativamente, eliminando los errores por timeout observados en SQL sin índices.

NoSQL como solución intermedia:

- Aunque NoSQL supera claramente a SQL sin índices en términos de latencia y estabilidad, no iguala el rendimiento de SQL con índices en operaciones GET. Sin embargo, para ciertas aplicaciones con acceso más distribuido y menos estructurado, NoSQL podría ser más adecuado.

POST (Escritura):

- En operaciones POST, NoSQL y SQL tienen un rendimiento comparable, aunque SQL con índices tiene una ligera ventaja en latencia y solicitudes por segundo.

Conclusiones finales

SQL con índices es el claro ganador:

- En términos de latencia, volumen de solicitudes procesadas y estabilidad bajo carga, SQL con índices se desempeña mejor en este análisis.
- Es la opción ideal para sistemas donde la velocidad y la capacidad de manejar grandes volúmenes de solicitudes son prioritarias.

NoSQL puede ser útil dependiendo del caso de uso:

- Aunque no alcanza los niveles de rendimiento de SQL con índices, NoSQL es una opción competitiva en escenarios donde los datos son menos estructurados o el sistema necesita flexibilidad adicional para manejar datos distribuidos.

SQL sin índices es ineficiente:

- Este escenario muestra tiempos de respuesta y volúmenes de procesamiento inaceptables bajo carga, con errores significativos que lo convierten en una opción no viable para producción.

Anexos

Load testing: https://docs.google.com/spreadsheets/d/1gMiQwqffgLYEuXyWnOrkMtU7VS-km-hhcN_jmlnfHGQ/edit?usp=sharing