Optimización de Arquitectura Backend en la Nube para un Rendimiento Rentable: Resolución de consultas N+1 y estrategias de caché con Redis

Andrés Felipe Calvo Ariza

Trabajo de grado para optar al título de Ingeniero de Sistemas

Asesor

Astrid Duque Ramos, Ph.D en Computer Science

Universidad de Antioquia

Facultad de Ingeniería

Ingeniería de Sistemas

Medellín

2026

TABLA DE CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

SIGLAS, ACRÓNIMOS Y ABREVIATURAS

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

APM: Application Performance Monitoring (Monitoreo de Rendimiento de Aplicaciones).

B-Tree: Balanced Tree (Estructura de árbol balanceado utilizada para la indexación de bases de datos).

N+1: Patrón de ineficiencia en consultas a bases de datos donde se ejecutan peticiones redundantes en ciclos iterativos.

ORM: Object-Relational Mapping (Técnica de programación para convertir datos entre sistemas incompatibles usando programación orientada a objetos).

PR: Pull Request (Solicitud de integración de cambios en un repositorio de código).

RAM: Random Access Memory (Memoria de acceso aleatorio utilizada por Redis para almacenamiento de alta velocidad).

RFC: Request for Comments (Documento técnico que describe una propuesta de mejora o estándar para ser revisado por pares).

S.A.S.: Sociedad por Acciones Simplificada.

SQL: Structured Query Language (Lenguaje de consulta estructurada para la gestión de bases de datos).

TTL: Time To Live (Tiempo de vida definido para los datos almacenados en la memoria caché).

RESUMEN

En la ingeniería de software contemporánea, la latencia representa el indicador fundamental de la salud y eficiencia de un sistema distribuido. En el contexto de Simetrik S.A.S., se identificaron tiempos de respuesta elevados en servicios de alto tráfico y flujos de seguridad que comprometían la escalabilidad y estabilidad de la plataforma. El objetivo central de este trabajo consistió en **optimizar el rendimiento del backend** para reducir la latencia y aumentar la eficiencia operativa del sistema.

La metodología se desarrolló bajo un marco de investigación aplicada estructurado en cinco fases, fundamentado en el diagnóstico técnico basado en evidencias y la observabilidad. El enfoque consistió en la caracterización de servicios críticos, la refactorización de la lógica de acceso a datos y la implementación de estrategias de persistencia temporal para mitigar la carga transaccional sobre el motor de base de datos principal.

Los resultados demostraron una optimización sustancial, destacando una **reducción de latencia de hasta el 78%** en servicios de monitoreo y una mejora del **68% en los percentiles de respuesta extrema (P99)** dentro de los flujos de autorización. El proyecto concluye que la intervención sistemática en la arquitectura de persistencia fortalece la robustez y previsibilidad del sistema, garantizando una infraestructura escalable capaz de manejar incrementos masivos en el volumen de datos.

Palabras clave **— Optimización de rendimiento, Consultas N+1, Carga ansiosa, Indexación de bases de datos, Persistencia temporal, Observabilidad de software, Escalabilidad de backend**

ABSTRACT

In contemporary software engineering, latency serves as the fundamental indicator of a distributed system's efficiency and health. Within the operational context of Simetrik S.A.S., elevated response times were identified in high-traffic services and security flows, which compromised platform scalability and stability. The primary objective of this work was to optimize backend performance to reduce response times and increase the system's operational efficiency.

The methodology was developed under an applied research framework structured into five phases, grounded in evidence-based technical diagnosis and observability. The approach focused on the characterization of critical services, the refactoring of data access logic, and the implementation of temporal persistence strategies to mitigate the transactional load on the primary database engine.

The results demonstrated substantial optimization, highlighted by a **latency reduction of up to 78%** in monitoring services and a **68% improvement in extreme response percentiles (P99)** within authorization flows. The project concludes that systematic architectural interventions strengthen system robustness and predictability, ensuring a scalable infrastructure capable of handling massive increases in data volume.

Keywords **— Performance optimization, N+1 queries, Eager Loading, Database indexing, Temporal persistence, Software observability, Backend scalability.**

I. INTRODUCCIÓN

En el ámbito de la ingeniería de software moderna, la latencia no debe interpretarse únicamente como un parámetro temporal elemental, sino como una variable crítica que define la integridad y el rendimiento de una infraestructura distribuida. En un entorno tecnológico donde la inmediatez es la norma, minimizar estos tiempos de respuesta es el desafío central de la arquitectura de sistemas, ya que cada milisegundo de retraso representa una fricción en el flujo de datos y una barrera crítica para la interactividad y la escalabilidad operativa.

Para organizaciones como Simetrik S.A.S., un bajo rendimiento en los servidores backend trasciende lo técnico y se convierte en un riesgo operativo y de negocio crítico. Dado que la compañía gestiona flujos masivos de datos y procesos de conciliación complejos, una latencia elevada impacta directamente en la productividad del cliente, quien depende de la velocidad de procesamiento para la toma de decisiones financieras en tiempo real. Un backend ineficiente no solo genera frustración en el usuario final, sino que aumenta los costos de infraestructura, reduce la capacidad de escala de la plataforma y compromete la confianza en la estabilidad del servicio, afectando la competitividad de la empresa en un mercado altamente exigente.

Históricamente, estos problemas de rendimiento no suelen originarse por un fallo único, sino por la acumulación de deuda técnica y cuellos de botella arquitectónicos que emergen a medida que los sistemas escalan. El uso de mapeadores objeto-relacionales (ORM), si bien acelera el desarrollo, suele introducir ineficiencias "silenciosas" como el patrón de consultas redundantes N+1. Esta evolución tecnológica ha llevado a la industria a reconocer que las aplicaciones modernas requieren una supervisión constante, teniendo en cuenta las ineficiencias que antes eran imperceptibles hoy se vuelven críticas bajo condiciones de alta carga y concurrencia.

En este contexto, las herramientas de monitoreo y observabilidad surgen como un aliado estratégico. El empleo de plataformas de gestión de rendimiento de aplicaciones y herramientas de depuración técnica permite inspeccionar la naturaleza interna del backend a través del análisis de trazas distribuidas. Una traza se define técnicamente como la representación del ciclo de vida completo de una petición a medida que transita por los diversos componentes de un sistema. A diferencia de los registros de eventos (logs) convencionales, la traza proporciona una vista jerárquica y cronológica compuesta por spans o segmentos, los cuales documentan el tiempo exacto de ejecución de cada operación individual: desde la recepción de la solicitud en el middleware hasta la ejecución de una consulta en la base de datos o una llamada a un servicio externo.

Estas trazas proporcionan una radiografía exacta del comportamiento sistémico, permitiendo identificar con precisión quirúrgica dónde se localizan los cuellos de botella. Su utilidad es determinante para detectar ineficiencias ocultas, como el patrón de consultas N+1, donde una traza revela visualmente una sucesión masiva de peticiones redundantes a la base de datos que saturan el canal de comunicación. Al transformar un flujo de ejecución opaco en una secuencia de eventos medibles, la observabilidad permite que la optimización deje de ser un proceso de ensayo y error para convertirse en una intervención basada en evidencias, garantizando que cada ajuste técnico tenga un impacto directo y cuantificable en la reducción de la latencia global.

El objetivo central de este proyecto es optimizar el rendimiento de dichos componentes mediante un proceso sistemático de análisis, diagnóstico y aplicación de mejoras técnicas orientadas a la reducción de tiempos de respuesta y al incremento de la eficiencia operativa. La metodología adoptada se ejecutó de forma estructurada en fases que iniciaron con un diagnóstico técnico detallado para detectar ineficiencias específicas, tales como la carencia de índices estratégicos en la capa de persistencia y el uso de estructuras de datos subóptimas para procesos de alta demanda. A partir de los hallazgos obtenidos, la fase de implementación se centró en la refactorización de la lógica de acceso a datos y en la integración de Redis como un sistema de caché distribuido para optimizar el flujo de autorización y mitigar la carga transaccional sobre el motor de base de datos principal. Finalmente, el impacto de estas intervenciones fue validado mediante una comparación exhaustiva de métricas de rendimiento, asegurando que las soluciones fortalecieran la robustez y escalabilidad del backend ante el crecimiento futuro de la plataforma.

II. OBJETIVOS

A. Objetivo general

Optimizar el rendimiento de los endpoints y del flujo de autenticación del backend de la empresa Simetrik S.A.S, mediante el análisis, diagnóstico y aplicación de mejoras orientadas a reducir la latencia y aumentar la eficiencia de las operaciones.

B. Objetivos específicos

1. Identificar los diez endpoints con mayor volumen de solicitudes y niveles de latencia.
2. Diagnosticar las causas técnicas que afectan el rendimiento en los endpoints priorizados.
3. Ejecutar optimizaciones en la capa de persistencia y en la lógica de las consultas.
4. Reestructurar el flujo de autenticación del sistema.
5. Formalizar la documentación técnica de las mejoras y cambios arquitectónicos realizados.
6. Cuantificar la variación en el desempeño del sistema tras la implementación de las mejoras.

III. MARCO TEÓRICO

El presente marco teórico profundiza en los fundamentos técnicos y conceptuales necesarios para abordar la optimización de sistemas backend de alto tráfico. Se centra en la reducción de latencia, la eficiencia de las consultas relacionales y la implementación de capas de persistencia temporal mediante una arquitectura de microservicios robusta.

A. Ingeniería de Rendimiento y Análisis Estadístico de Latencia

En la computación distribuida contemporánea, el rendimiento no debe entenderse como una métrica determinista y estática, sino como una distribución de probabilidad que fluctúa dinámicamente según la carga de trabajo y la contención de recursos. Técnicamente, la latencia se define como el intervalo de tiempo transcurrido desde que se emite una solicitud hasta que se recibe la respuesta completa por parte del cliente. Para gestionar este indicador, las investigaciones fundamentales de Dean y Barroso [1] demuestran que, en sistemas a gran escala, el uso de la media aritmética es insuficiente y a menudo engañoso. Este promedio tiende a ocultar variaciones críticas provocadas por el ruido sistémico, tales como la interferencia de procesos externos, las pausas por recolección de basura (Garbage Collection) o fallos parciales en la red, los cuales impactan solo a una fracción de las peticiones, pero pueden degradar severamente la percepción global del servicio.

En consecuencia, la ingeniería de rendimiento propone el uso de medidas de posición estadística conocidas como percentiles para obtener una visión granular y fiel de la realidad operativa. Mientras que el P50 o mediana describe la experiencia del usuario promedio, indicadores como el P95 y el P99 resultan determinantes para evaluar la verdadera escalabilidad y robustez de la infraestructura. La importancia crítica de estos percentiles superiores radica en que representan los escenarios de borde o valores atípicos que revelan las ineficiencias estructurales más profundas del sistema. Un P99 elevado es un síntoma inequívoco de falta de predictibilidad; indica que, bajo condiciones de estrés, una parte de los usuarios experimenta tiempos de respuesta que pueden ser órdenes de magnitud superiores a la mediana, lo cual precede habitualmente a saturaciones de recursos o fallos en cascada.

Este fenómeno se conoce técnicamente como el problema de la cola larga (Long Tail Latency), el cual establece que la latencia total de una operación compleja está dictada por el componente más lento de la cadena de ejecución. En arquitecturas de microservicios, donde una sola petición del usuario a menudo desencadena múltiples llamadas paralelas a diversos servicios

o bases de datos, si tan solo una de esas sub-operaciones cae en el extremo lento de la distribución, toda la transacción principal se verá retrasada para el usuario final. Por tanto, un percentil 99 elevado indica que el sistema carece de predictibilidad bajo condiciones de estrés, lo cual compromete la estabilidad global de la plataforma. Para comprender el origen de estas variaciones y localizar con precisión los componentes responsables de esta latencia residual, es necesario trascender la métrica externa e inspeccionar el estado interno del sistema a través de la observabilidad.

B. Los Pilares de la Observabilidad en Sistemas Complejos

Con el objetivo de mitigar los efectos de la cola larga, se requiere una visibilidad profunda que el monitoreo convencional basado en umbrales estáticos no puede ofrecer por sí solo. En este contexto, la observabilidad se define como la propiedad de un sistema que permite inferir su estado interno únicamente a partir del análisis de sus señales externas o telemetría. A diferencia del monitoreo tradicional, la observabilidad proporciona la profundidad necesaria para diagnosticar comportamientos imprevistos conocidos como incógnitas desconocidas dentro de las infraestructuras distribuidas [2]. Esta capacidad es determinante en sistemas modernos donde la complejidad de las interacciones impide prever todos los modos de fallo posibles.

Para alcanzar este nivel de transparencia operativa, Sigelman et al. [2] proponen la integración de tres fuentes de datos fundamentales: métricas, que ofrecen una visión cuantitativa de la salud del sistema; logs, que proporcionan el contexto textual de las transacciones; y trazas distribuidas, que representan el flujo causal de una petición. Cada traza se compone de unidades de trabajo denominadas “Spans”, las cuales documentan de forma jerárquica un tiempo de inicio y fin determinado. El análisis de estos segmentos a través de gráficos de flama permite identificar visualmente cuellos de botella críticos, evidenciando el tiempo de ejecución bloqueante frente al tiempo de espera asíncrono en cada operación técnica [3].

Esta capacidad de inspección granular permite confirmar que, en servicios de alto tráfico, la degradación del rendimiento suele originarse en la forma en que el software interactúa con la capa de datos. Al analizar las trazas de ejecución, se hace evidente que el uso de abstracciones para la gestión de persistencia puede introducir ineficiencias severas, lo que conduce necesariamente al estudio de los mapeadores objeto relacionales y sus patrones de consulta redundantes.

C. Optimización de la Persistencia y el Desajuste de Impedancia (ORM)

Al realizar una inspección detallada, se identifica frecuentemente que el origen de los retardos sistémicos reside en la comunicación entre el código y la base de datos. Los Object Relational Mappers (ORM) actúan como un puente técnico simplificando la manipulación de información; no obstante, esta capa de abstracción introduce el fenómeno conocido como desajuste de impedancia, en el cual la estructura lógica del código de aplicación no se traduce eficientemente a sentencias SQL optimizadas [4]. Este desajuste surge de la diferencia entre el modelo de objetos y el relacional, generando consultas innecesariamente complejas que ralentizan el sistema.

La manifestación más crítica de este desajuste es el patrón de consultas N+1, descrito por Silva et al. [4] como una ineficiencia donde el motor de aplicaciones realiza una consulta inicial y, posteriormente, emite una adicional por cada registro para obtener sus datos relacionados. Técnicamente, esto incrementa el costo computacional de O(1) a una progresión lineal de O(n), saturando el canal de comunicación a través de protocolos TCP IP con micro peticiones redundantes. La acumulación de estas llamadas aumenta drásticamente la latencia percibida, transformando procesos simples en cuellos de botella que comprometen la escalabilidad.

Para mitigar este problema, se emplean estrategias de carga anticipada o Eager Loading. Una de ellas es el Join based Loading (select_related), el cual consolida la recuperación de información en una única operación de JOIN, siendo óptimo para relaciones de cardinalidad uno a uno o muchos a uno. Por otro lado, el Batch based Loading (prefetch_related) ejecuta consultas separadas y realiza la unión semántica en la memoria del servidor, evitando productos cartesianos masivos que degradarían la memoria del motor relacional [4]. Sin embargo, la eficiencia final de estas consultas, por optimizadas que estén en el código, depende de la capacidad del motor para localizar la información físicamente, lo que resalta la importancia de la estructura de datos en disco.

D. Teoría de Indexación y Estructuras de Datos en Disco

Más allá de la eficiencia en el número de peticiones generadas por el software, la velocidad de respuesta final está condicionada por la arquitectura física de almacenamiento. La recuperación eficiente en tablas de alta densidad depende de evitar el escaneo secuencial o Sequential Scan, un proceso de búsqueda con costo lineal O(n) que obliga al motor a leer cada página de datos en disco hasta encontrar la coincidencia, degradando el rendimiento general en entornos de alta concurrencia.

Para resolver este problema estructural, Bayer y McCreight [5] propusieron el uso del índice B-Tree, una estructura que organiza las claves de forma jerárquica y balanceada. Esta organización permite localizar cualquier registro siguiendo un camino de punteros con una complejidad logarítmica O(log n). En términos operativos, buscar un registro entre un millón de filas requiere aproximadamente veinte comparaciones en un B-Tree, frente al millón de lecturas de un escaneo secuencial. La ausencia de estos índices es la causa principal de latencias extremas en el percentil 99 [5], ya que obliga al sistema a realizar esfuerzos computacionales desproporcionados.

No obstante, incluso con una indexación estratégica, el acceso constante a medios de persistencia física sigue limitado por latencias mecánicas y de hardware. Esta barrera física introduce la necesidad de implementar capas de persistencia temporal en memoria para servir datos frecuentes de forma casi instantánea, lo cual conduce al análisis de las arquitecturas de caché distribuida.

E. Arquitectura de Caché y Persistencia Temporal (Redis)

Incluso con una organización óptima de datos en disco, la persistencia física impone barreras de latencia que solo pueden superarse mediante el desplazamiento de la carga hacia la memoria volátil. El almacenamiento en caché explota el principio de localidad de referencia para servir datos frecuentes desde la memoria RAM, reduciendo drásticamente la carga transaccional del motor principal y evitando que consultas repetitivas saturen los recursos de entrada y salida [6].

En este ámbito, sistemas como Redis [7] operan como almacenes de datos clave valor que ofrecen latencias sub-milisegundo. Wang et al. [6] explican que, al descentralizar datos de alta frecuencia, tales como flujos de autorización y permisos, hacia una instancia en memoria, se mitiga la contención de recursos en el motor relacional. Este desacoplamiento protege la base de datos primaria ante ráfagas de tráfico y permite una mayor concurrencia operativa al distribuir la carga entre capas de persistencia diferenciadas.

Para garantizar la fiabilidad de esta capa temporal y evitar datos obsoletos, es indispensable la implementación de políticas de expiración o Time To Live (TTL). Esta configuración asegura que el caché se invalide automáticamente tras un periodo definido, forzando una actualización desde la fuente de verdad. Dicha gestión de consistencia es crítica en procesos de seguridad, donde los cambios en privilegios deben propagarse en un tiempo predecible para mantener la integridad de la plataforma [6]

IV. METODOLOGÍA

El presente proyecto se desarrolla bajo un marco de investigación aplicada con un enfoque en la optimización iterativa de sistemas distribuidos. La metodología se fundamenta en la observabilidad y el diagnóstico basado en evidencias, permitiendo que cada intervención técnica responda a métricas precisas recolectadas en entornos reales de ejecución.

A. Instrumentos y herramientas de diagnóstico

La fase de auditoría y validación se fundamentó en un ecosistema de herramientas especializadas, seleccionadas por su capacidad para proporcionar visibilidad granular sobre el comportamiento del backend y la eficiencia de la capa de persistencia. Como eje central de la observabilidad, se empleó Datadog APM [8] para la recolección de métricas de latencia y el análisis de trazas distribuidas, lo que permitió localizar cuellos de botella sistémicos y establecer una línea base objetiva mediante la caracterización de percentiles.

Complementariamente, en el ciclo de desarrollo local se utilizó Django Debug Toolbar [9] para realizar auditorías profundas de las sentencias SQL generadas por el ORM, detectando de forma temprana patrones de consultas N+1. Para la optimización de la persistencia, se integró Redis como motor de caché en memoria de baja latencia, mitigando la carga transaccional sobre la base de datos relacional principal en los flujos de autorización.

Finalmente, para mitigar riesgos operativos, se utilizó copias de seguridad de las bases de datos en ambientes de desarrollo; esto garantizó que las pruebas de carga y la validación de índices se ejecutarán en condiciones fieles a la realidad productiva sin comprometer la integridad del sistema.

B. Fases del proyecto

1. **Fase I: Identificación y establecimiento de la línea base:** Se analizó el tráfico transaccional mediante Datadog APM para seleccionar los diez servicios con mayor criticidad. Como criterio de selección se priorizaron endpoints con alto volumen de solicitudes, dada su integración directa en flujos críticos del frontend y su impacto masivo en la experiencia de usuario. Asimismo, se seleccionaron aquellos servicios cuya latencia presentara un percentil P99 cercano o superior a 300ms.

2. **Fase II: Diagnóstico técnico y análisis de trazas:** Se realizó un estudio detallado de cada servicio crítico para detectar ineficiencias como consultas N+1, escaneos secuenciales de tablas por falta de índices y sobrecargas en el middleware de autenticación. Se utilizaron planes de ejecución de base de datos para validar el costo computacional de cada sentencia.
3. **Fase III: Optimización e implementación:** Esta fase comprendió la refactorización de código, la optimización de esquemas de indexación y la integración de estrategias de caché distribuida. Cada solución se documentó inicialmente mediante un RFC (Request for Comments), detallando el contexto técnico y la justificación arquitectónica del cambio.
4. **Fase IV: Validación arquitectónica y revisión por pares:** Los RFCs fueron sometidos a la auditoría técnica del equipo de ingeniería. Una vez aprobados, se procedió a la creación de Pull Requests (PRs) para integrar formalmente las mejoras en la rama de desarrollo mediante procesos de integración continua.
5. **Fase V: Validación de impacto y contraste de métricas:** Finalmente, se midió el impacto real de las optimizaciones comparando las métricas de latencia (P50 a P99) recolectadas antes y después de las intervenciones. Este contraste permitió validar la estabilidad del sistema frente a picos de tráfico y condiciones de borde

V. RESULTADOS

La evaluación de las mejoras técnicas se fundamentó en la selección de los percentiles de latencia (*P50*, *P75*, *P90*, *P95*, *P99*) como métricas fundamentales debido a su capacidad para ofrecer una visión granulada de la distribución del rendimiento, superando las limitaciones de los promedios simples que suelen ocultar variaciones críticas en el comportamiento del sistema. Estos indicadores, actuando como medidas de posición estadística, permitieron determinar con precisión el valor de latencia por debajo del cual se situó un porcentaje específico de las solicitudes procesadas, facilitando la identificación del impacto de las optimizaciones en diversos segmentos de usuarios. A continuación se describe la relevancia operativa y el propósito analítico de los percentiles seleccionados dentro de este estudio:

- **P50 (Mediana):** Representa la experiencia estándar percibida habitualmente por el usuario.
- **P75 y P90:** Utilizados como métricas de referencia para capturar el rendimiento en usuarios con tiempos de espera superiores al promedio.
- **P95 y P99:** Indicadores críticos empleados para evaluar la estabilidad del sistema ante escenarios de alta demanda y valores atípicos.

A partir de los criterios de selección y priorización técnica establecidos en el capítulo de metodología, se consolidó un conjunto de diez endpoints críticos para la fase de evaluación y optimización. La Tabla 1 resume la carga transaccional y la densidad de tráfico capturada por Datadog APM durante el periodo de observación, validando el alto impacto operativo que estos servicios representan para la estabilidad de la plataforma.

TABLA I
VOLUMEN DE SOLICITUDES Y P99 POR ENDPOINT CRÍTICO (JULIO-AGOSTO 2025)

Endpoint	Peticiones en miles	Latencia P99 (ms)
GET rule-key-columns	141	1100
GET reconciliations/custom-formats	582	538
GET reconciliations/sources/id/segments	497	1000
GET reconciliations/columns	867	722
GET task_monitor/events	37.7	7160
GET resource-origin-columns	190	581
GET generated_files	66	648
GET authorization	11400	413
GET v2/users/grouped-permissions	302	870
POST v2/authorize	10200	272

A. Evaluación de Resultados y Comparativa de Rendimiento

Se realizó un análisis comparativo entre el estado previo y posterior a la implementación de las optimizaciones, con el fin de validar la efectividad de las intervenciones técnicas. El estudio se centró en los diez endpoints críticos identificados inicialmente, utilizando Datadog APM como fuente de datos para extraer los tiempos de respuesta en los percentiles P50, P75, P90, P95 y P99.

A continuación, se presenta la tabla comparativa donde se detalla el comportamiento del sistema antes y después de la resolución de consultas N+1, la optimización de índices SQL y la integración de Redis como sistema de caché.

TABLA II
COMPARATIVA DE LATENCIAS (ms) POR PERCENTIL ANTES Y DESPUÉS DE LA INTERVENCIÓN

Endpoint	Periodo	P50	P75	P90	P95	P99
GET rule-key-columns	Antes	226	317	490	638	1100
	Después	146	171	226	308	581
GET reconciliations/custom-formats	Antes	166	193	260	343	538
	Después	129	153	203	285	581
GET reconciliations/sources/id/segments	Antes	163	263	327	440	1000
	Después	163	209	289	382	711
GET reconciliations/columns	Antes	240	327	388	468	722
	Después	303	317	338	365	433
GET task_monitor/events	Antes	1980	2610	3400	4160	7160
	Después	426	572	1030	1280	2570
GET resource-origin-columns	Antes	153	193	276	348	581
	Después	137	160	209	294	555
GET generated_files	Antes	268	317	382	447	648
	Después	127	153	196	264	498
GET authorization	Antes	30	52	106	171	413
	Después	23	30	45	66	158
GET v2/users/grouped-permissions	Antes	72	280	420	483	870
	Después	63	176	348	440	700
POST v2/authorize	Antes	34	57	89	125	272
	Después	23	27	44	58	86

TABLA III
IMPACTO PORCENTUAL (Δ %) EN LA REDUCCIÓN DE LATENCIA

Endpoint	P50 Δ%	P75 Δ%	P90 Δ%	P95 Δ%	P99 Δ%
GET rule-key-columns	-35	-46	-54	-52	-47
GET reconciliations/custom-formats	-22	-21	-22	-17	8
GET reconciliations/sources/id/segments	0	-21	-12	-13	-29
GET reconciliations/columns	26	-3	-13	-22	-40
GET task_monitor/events	-78	-78	-70	-69	-64
GET resource-origin-columns	-10	-17	-24	-16	-4
GET generated_files	-53	-52	-49	-41	-23
GET authorization	-23	-42	-58	-61	-62
GET v2/users/grouped-permissions	-13	-37	-17	-9	-20
POST v2/authorize	-32	-53	-51	-54	-68

Como se evidencia en los resultados, las optimizaciones lograron una reducción significativa, destacando una mejora del 68% en el *P99* del flujo de autorización y una reducción del 78% en el *P75* del servicio task_monitor. Estos valores confirman que las soluciones no solo aceleraron el tiempo de respuesta promedio, sino que fortalecieron la estabilidad del sistema frente a valores atípicos.

VI. ANÁLISIS

El análisis de los resultados se realizó mediante una metodología de diagnóstico individual para cada uno de los endpoints seleccionados. El objetivo principal fue validar la resolución de las ineficiencias técnicas comparando las trazas de ejecución antes y después de las intervenciones. Para ello, se ejecutó un flujo de trabajo estructurado en las siguientes cuatro etapas:

1. **Diagnóstico de trazas:** Localización de cuellos de botella mediante Datadog APM.
2. **Identificación de ineficiencias:** Detección de patrones como consultas ***N+1***, ausencia de índices o estructuras de datos lentas.
3. **Implementación de soluciones:** Ejecución de refactorizaciones (*select_related*, *prefetch_related*, *defer*) e integración de **Redis**.
4. **Validación de ejecución:** Análisis de métricas post-intervención para cuantificar la mejora en rendimiento y estabilidad.

Con el propósito de fundamentar este proceso de análisis, es necesario comprender la arquitectura distribuida del sistema y el flujo de telemetría que permitió la captura de evidencias. Como se detalla en el diagrama de arquitectura descrito en la Fig 1, el flujo de solicitudes es gestionado inicialmente por un balanceador de carga de Amazon Web Services [10], el cual dirige el tráfico hacia el componente de backend de conciliación ejecutándose sobre clusters de Amazon ECS [11]. Este servicio realiza operaciones críticas sobre una base de datos PostgreSQL [12], mientras delega las validaciones de seguridad al servicio de autorización.

Este último componente optimiza su rendimiento operativo mediante la interacción con un cluster de Redis para la gestión de persistencia temporal y una base de datos PostgreSQL adicional para el almacenamiento persistente. La visibilidad completa del sistema, indispensable para la localización de los cuellos de botella descritos en la etapa de diagnóstico, se logra a través de agentes de Datadog desplegados en cada cluster. Estos agentes centralizan la recolección de métricas y trazas distribuidas, enviándolas a la plataforma de Datadog en la nube para su posterior análisis y validación.

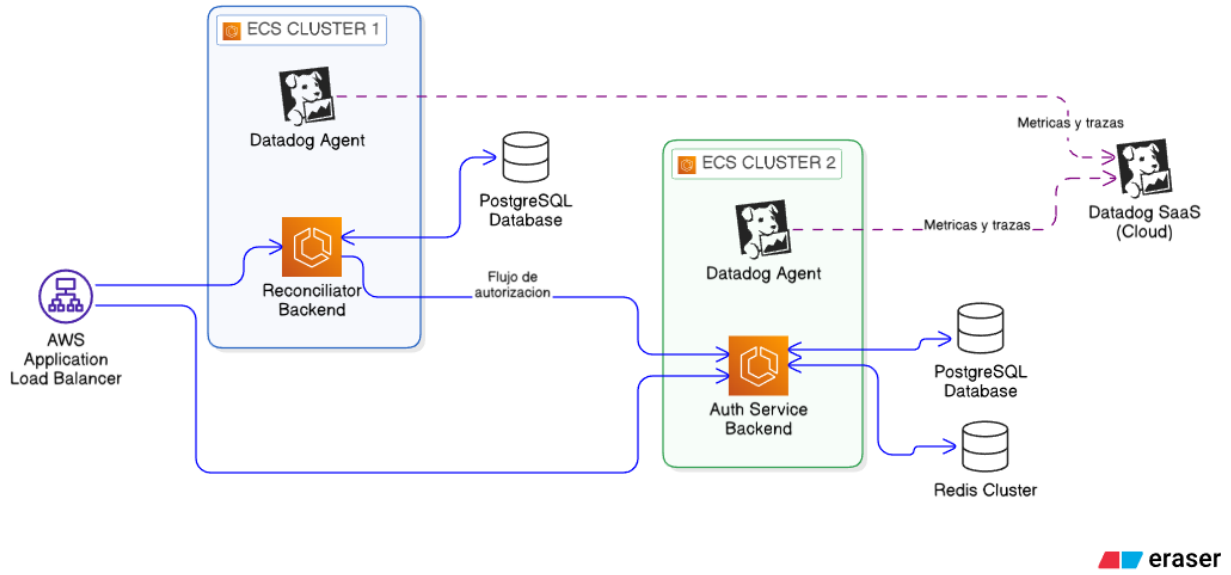

Fig.1. Arquitectura de servicios y ecosistema de observabilidad distribuida. Ilustración realizada con eraser.io [13]

A. Optimización de capa de persistencias y consultas relacionales

La principal causa de degradación del rendimiento en el backend fue la presencia del patrón de consultas N+1, el cual se manifiesta cuando el sistema realiza múltiples peticiones individuales a la base de datos dentro de un ciclo iterativo en lugar de consolidarlas en una única operación.

Para detectar este patrón, se realizó un análisis de trazas distribuidas utilizando el endpoint rule-key-columns como caso de referencia. Como se ilustra en el modelo relacional de la Fig. 2, la entidad Rule mantiene una dependencia con la entidad Column a través de dos llaves foráneas denominadas column_a y column_b. Esta estructura, aunque normalizada y consistente a nivel de base de datos, se convierte en un riesgo de rendimiento si el motor de persistencia (ORM) no gestiona adecuadamente la recuperación de las relaciones durante la serialización de los datos.

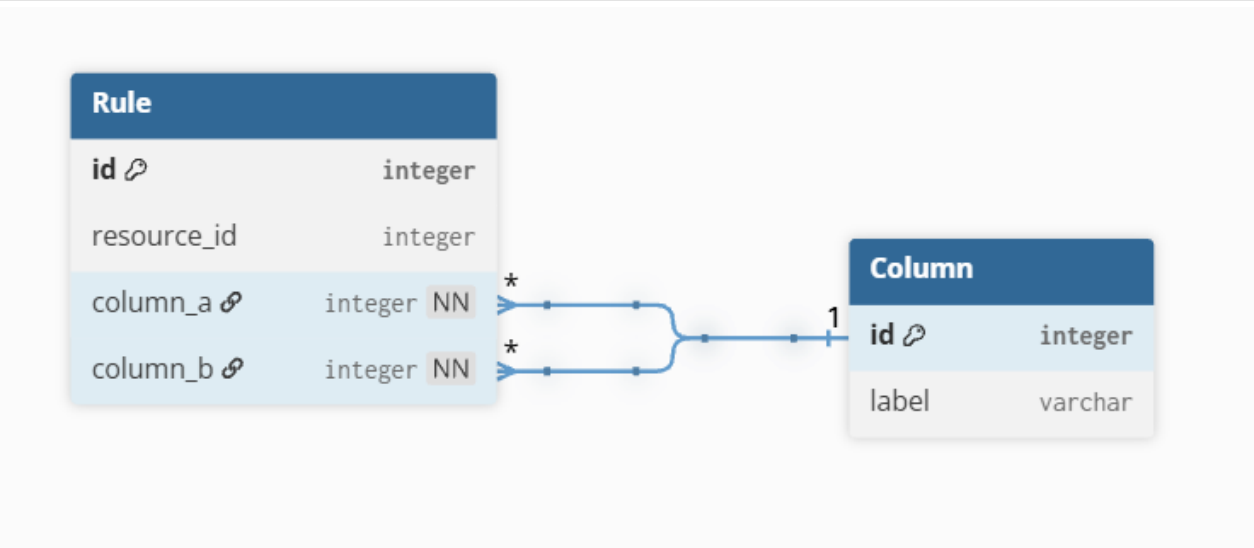

Fig 2. Modelo de Entidad-Relación que evidencia la dependencia doble entre Rule y Column. Ilustración realizada con dbdiagram.io [14]

Al investigar el origen del problema, la auditoría del código reveló una implementación que ignoraba las capacidades de carga ansiosa del framework, optando por una recuperación ineficiente de los datos. El fragmento de código de la Fig. 3 muestra el patrón típico que dispara un N+1, donde se debe prestar especial atención al acceso de atributos de objetos relacionados dentro de bucles. En las líneas donde se definen los conjuntos rule _columns, el código accede repetidamente a rule.column_a.id; aunque técnicamente sólo se requiere el identificador numérico, el ORM de Django [15], al carecer de una instrucción de unión previa como select_related o una proyección mediante .values(), se ve obligado a realizar una consulta SQL independiente por cada objeto Rule procesado para instanciar la entidad relacionada antes de extraer su atributo.

```python
def get_rule_columns(resource_id: number) -> set[int]:

    rule_columns: set[int] = set()
    rules: list[Rule] = Rule.objects.filter(
        resource_id=resource_id
    )

    rule_columns |= {rule.column_a.id for rule in rules}
    rule_columns |= {rule.column_b.id for rule in rules}
    return rule_columns
```

Fig 3. Implementación original en el backend que genera consultas redundantes mediante el acceso iterativo a llaves foráneas. Ilustración realizada con carbon.now.sh [16]

Esta deficiencia en la capa de aplicación se traduce directamente en la traza de ejecución analizada en la Fig. 4. La correlación entre el código ineficiente y la observabilidad se confirma mediante tres marcadores críticos identificados durante la fase de diagnóstico. El primero es la cantidad excesiva de operaciones, identificado con el marcador (1), donde el código generó **948 operaciones** para atender una sola solicitud. Esta ineficiencia se manifiesta visualmente en la secuencialidad de las micro-consultas dentro del área delimitada por el marcador representado en el rectángulo (2), donde el Flame Graph revela una sucesión de llamados a la base de datos de micro-duración que **saturan el hilo de ejecución** al procesarse de forma estrictamente secuencial.

Finalmente, el síntoma definitivo de un sistema operando fuera de sus parámetros de eficiencia se confirma mediante el panel identificado con el número (3), el cual detalla el desbalance en la distribución de tiempo de ejecución. Se observa que la base de datos (product_db) consume el 59.5% del tiempo total de la transacción, superando significativamente al procesamiento interno del backend, que solo representa el 38.3%. Este desequilibrio operativo identifica un punto crítico de ineficiencia que sustenta la necesidad de una intervención técnica.

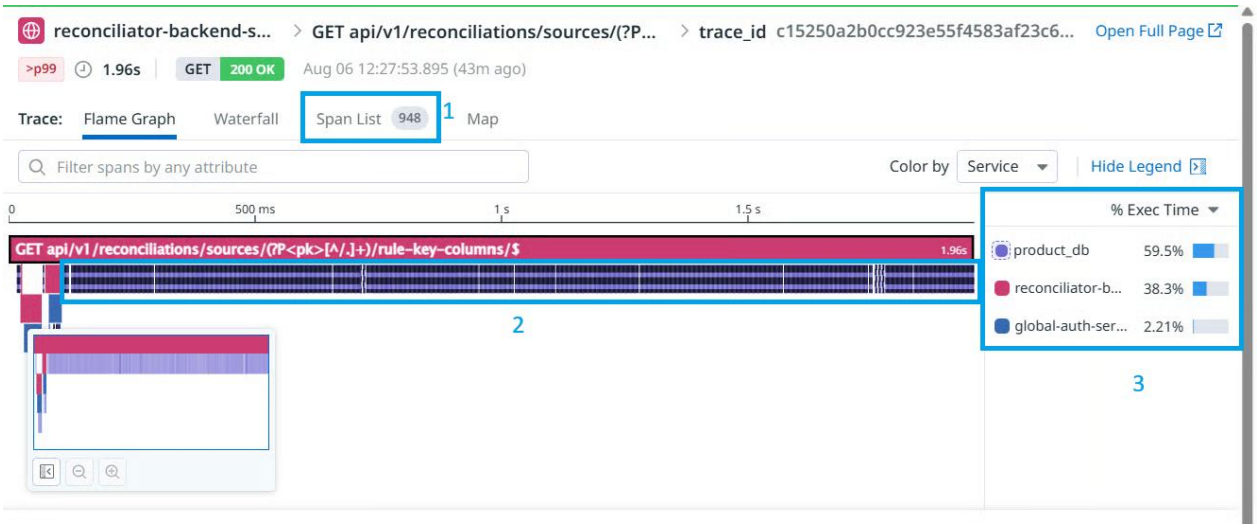

Fig. 4. Análisis de traza en Datadog APM para el endpoint rule-key-columns con indicadores de N+1 (1, 2, 3)

La solución aplicada consistió en una proyección de campos mediante .values('column_a_id', 'column_b_id'), lo que permitió extraer los IDs directamente de la tabla principal sin instanciar objetos relacionados, reduciendo la latencia del percentil 99 en un 47%.

```python
def get_rule_columns(resource_id: number) -> set[int]:

    rule_columns: set[int] = set()
    rules: list[Rule] = Rule.objects.filter(
        resource_id=resource_id
    ).values('column_a_id', 'column_b_id')

    for rule in rules:
        if rule['column_a_id']:
            rule_columns.add(rule['column_a_id'])
        if rule['column_b_id']:
            rule_columns.add(rule['column_b_id'])
    return rule_columns
```

Fig. 5. Implementación optimizada en el backend. Realizado con carbon.now.sh

Por otro lado, el uso de carga ansiosa (Eager Loading) mediante **select_related** fue la estrategia determinante en los servicios *custom-formats*, *reconciliations/columns* y *generated-files*. Por ejemplo, en el endpoint *generated-files*, la eliminación de consultas redundantes para obtener correos de usuario permitió reducir la mediana (*P50*) en un 52.6%. En el caso de *reconciliations/columns*, se logró una reducción del 40% en el *P99*, garantizando una estabilidad operativa crítica ante valores atípicos de la distribución.

Finalmente, en el servicio de segmentos, se detectó una ineficiencia por herencia de código que omitía configuraciones de prefetch_related. La solución integró carga ansiosa, diferencial y diferida mediante *defer('resource__folder')*, logrando contraer la cola larga (*P99*) en un 29% al eliminar el procesamiento de datos irrelevantes para el contexto de la interfaz.

B. Eficiencia en la persistencia mediante indexación y escaneo de tablas

El análisis de observabilidad en el servicio *task_monitor/events* reveló el caso de latencia más extremo del proyecto, con un *P99* de 7160 ms. El diagnóstico identificó que la ausencia de un índice en el campo *started_at* obligaba al motor de base de datos a realizar un escaneo secuencial (Sequential Scan) sobre la totalidad de la tabla.

La intervención técnica se centró en la creación concurrente de un **índice B-Tree**, logrando la mejora porcentual más significativa registrada: una reducción superior al 64% en todos los percentiles analizados. Este resultado valida que, en tablas con alta densidad de registros, la indexación estratégica es superior a cualquier refactorización de código a nivel de aplicación para resolver cuellos de botella estructurales.

C. Arquitectura de cache distribuida y comunicación backend-to-backend

El flujo de autorización de Simetrik S.A.S., con un volumen operativo de aproximadamente 10 millones de solicitudes mensuales, representaba históricamente la carga más crítica para el motor de persistencia relacional. En el diseño original, cada petición del usuario obligaba al backend a ejecutar consultas SQL secuenciales y dependientes: la validación de identidad, la obtención de permisos y la lógica de negocio respectiva. Como se ilustra en la Fig. 6, esta arquitectura generaba una latencia acumulada significativa debido a los múltiples viajes de red

(round-trips) y al procesamiento en disco necesario para resolver cada sentencia, creando un cuello de botella que comprometía la agilidad del sistema.

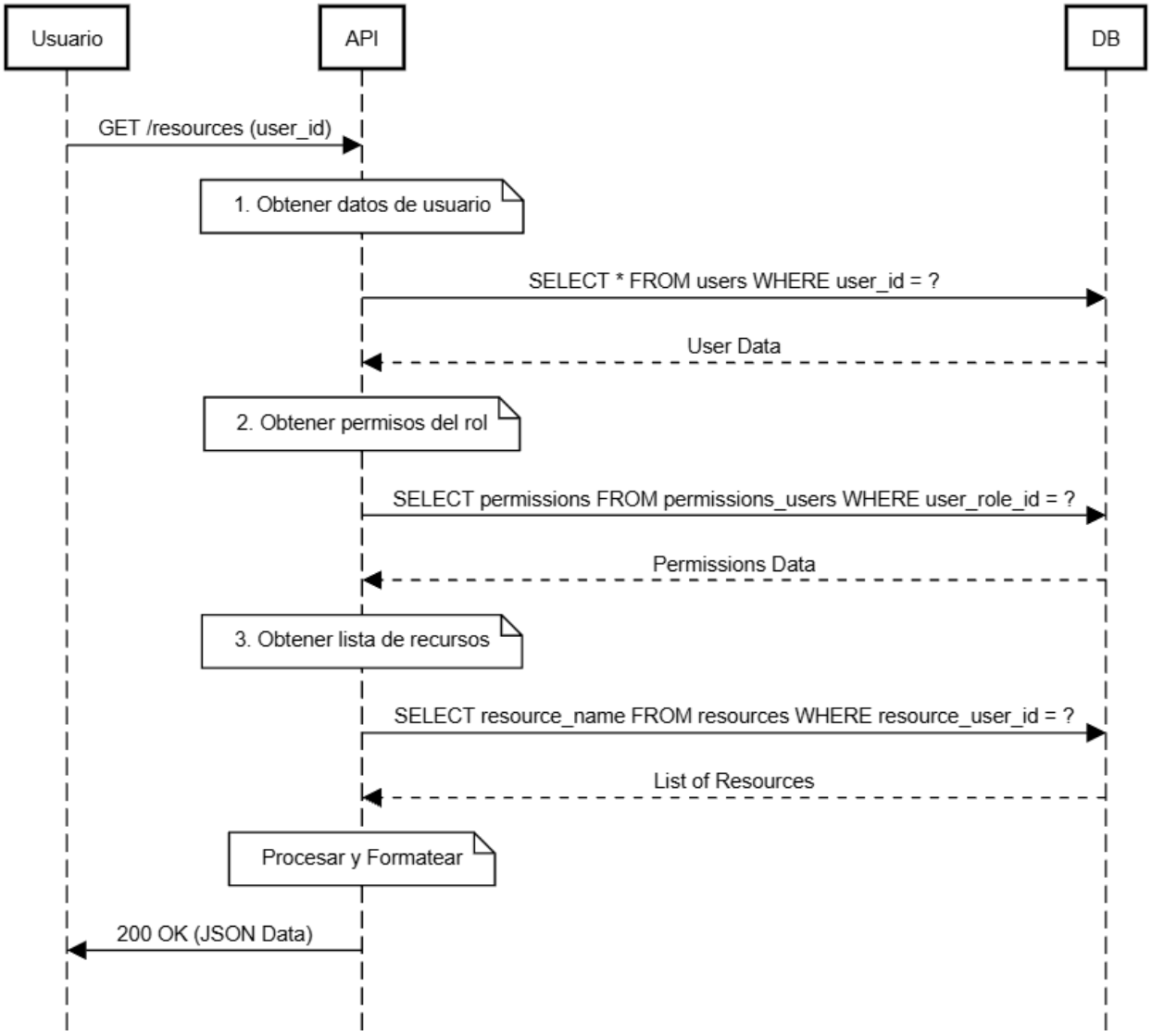

Fig 6. Diagrama de flujo secuencial sin Redis

Con el objetivo de descentralizar esta carga transaccional, se implementó una arquitectura de persistencia temporal utilizando Redis como sistema de caché distribuida. Como se detalla en el flujo de la Fig. 7, la estrategia establece que durante la primera interacción (escenario de cache ausente), el sistema guarda la información en Redis con tiempos de vida (TTL) diferenciados: 5 minutos para la información básica de sesión y 1 hora para la matriz de permisos. Esta segmentación asegura que los datos con menor tasa de cambio permanezcan disponibles por más

tiempo, optimizando el uso de la memoria y garantizando que el sistema solo consulte la base de datos para refrescar información estrictamente necesaria.

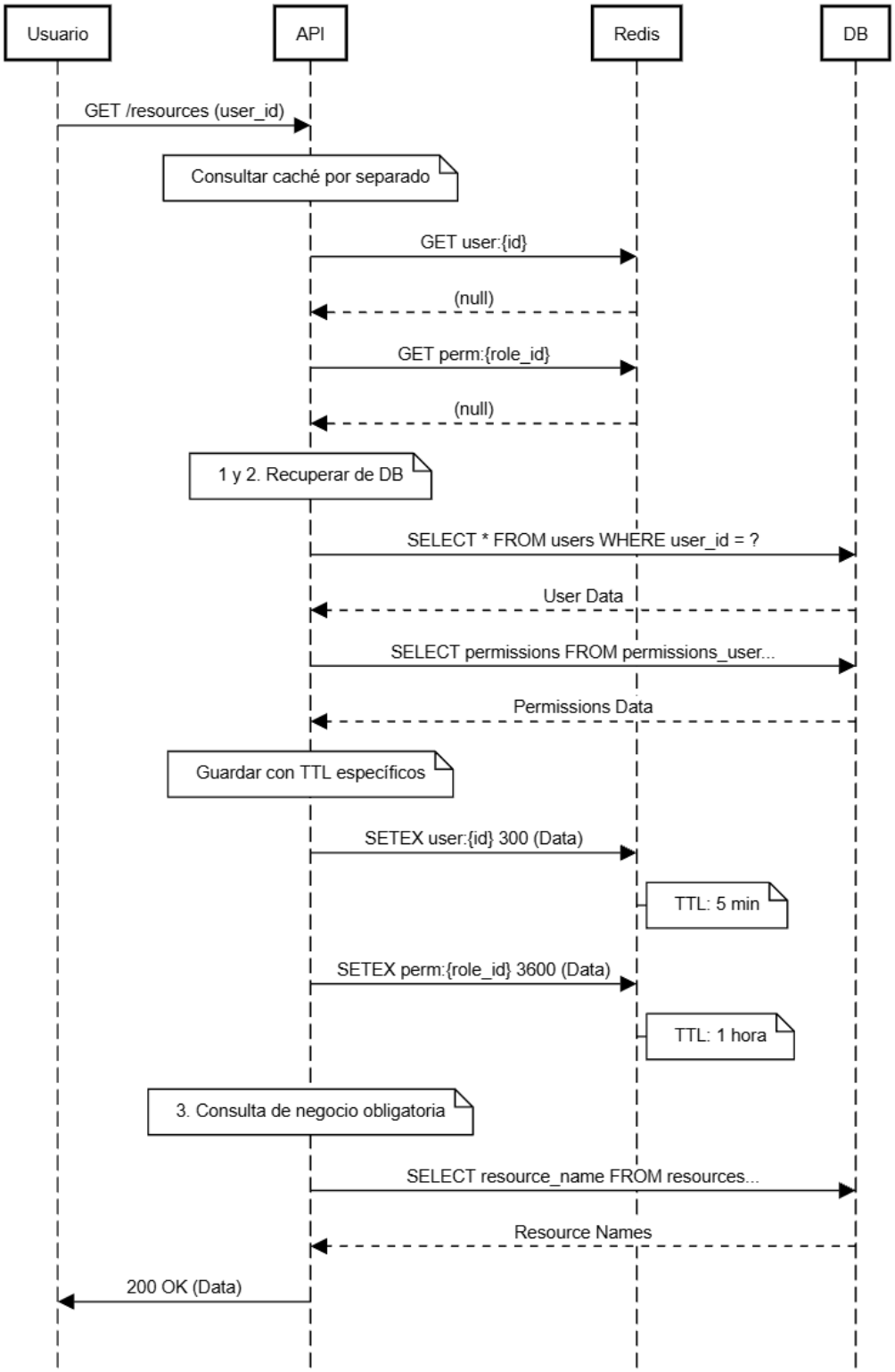

Fig 7. Diagrama de flujo para la petición inicial con Redis implementado

La efectividad de esta optimización se manifiesta plenamente en las peticiones subsecuentes, representadas en la Fig. 8, donde el backend recupera el contexto de identidad y permisos directamente desde Redis. Al omitir las consultas de autenticación en la base de datos principal, se permite la reutilización de estructuras de datos en memoria para servicios críticos como POST v2/authorize, logrando validaciones de alta velocidad sin incurrir en procesamiento relacional adicional. El impacto final de esta intervención se refleja en una mejora del 68% en el percentil P99 del flujo de autorización, reduciendo los tiempos de respuesta de 272 ms a 86 ms, lo cual fortalece la estabilidad sistémica y prepara al backend para manejar ráfagas de tráfico masivo de manera predecible.

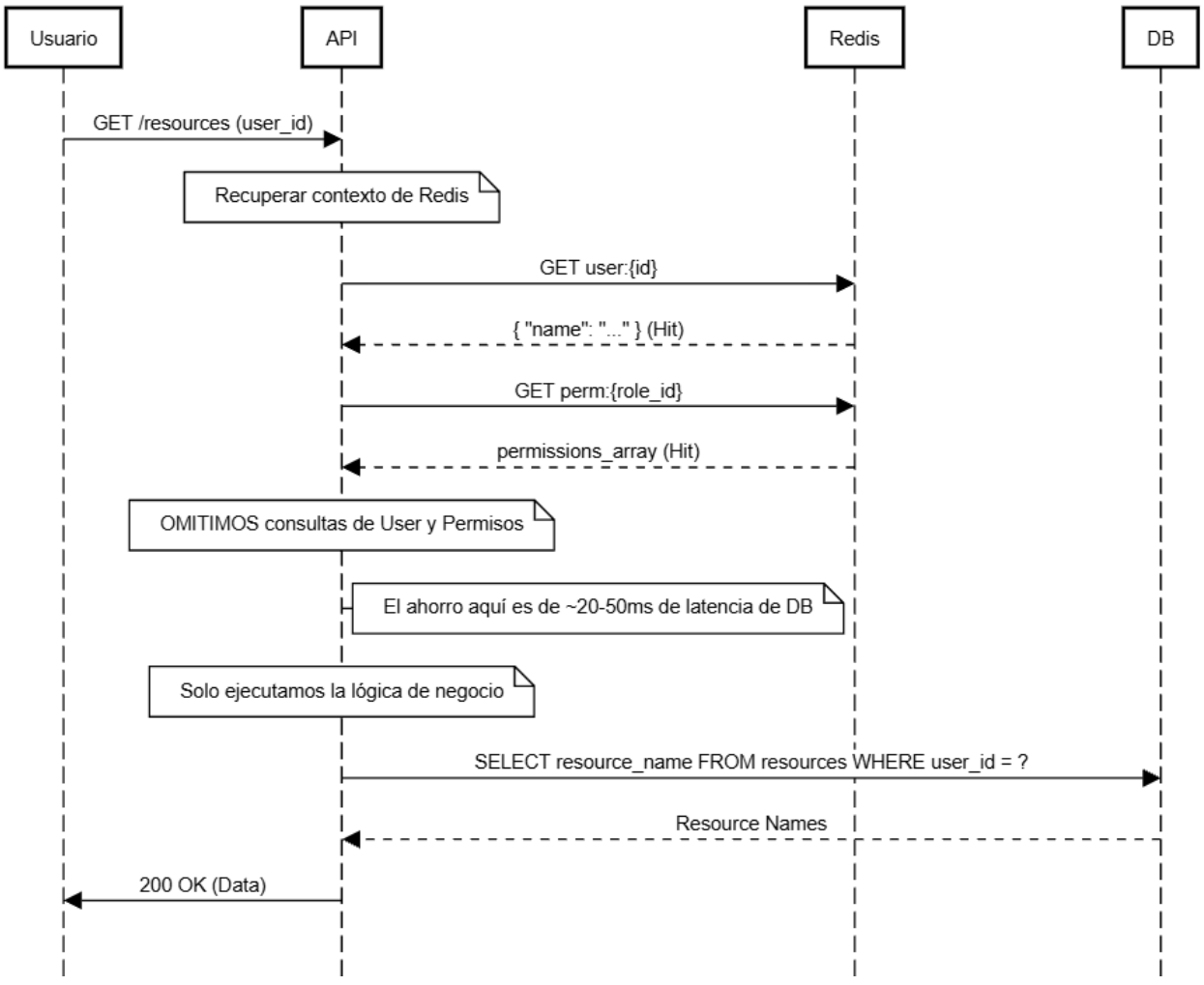

Fig 8. Diagrama de flujo de petición optimizada con cache hit

VII. CONCLUSIONES

A partir del desarrollo del proyecto de optimización, se concluye que el cumplimiento del objetivo general se alcanzó mediante un proceso sistemático de diagnóstico y refactorización que permitió reducciones de latencia de hasta el 78% en servicios críticos. Este éxito valida que un análisis basado en la observabilidad y el uso de métricas de alta fidelidad es la estrategia más efectiva para garantizar la escalabilidad de plataformas con alta densidad de tráfico. En este sentido, la eficacia de las refactorizaciones sobre el motor ORM de Django fue determinante, ya que la resolución del patrón de consultas N+1 mediante carga ansiosa y proyecciones de campos permitió transformar complejidades lineales O(n) en constantes O(1). Estas intervenciones no solo aceleraron el tiempo de respuesta promedio, sino que neutralizaron la degradación del rendimiento asociada al crecimiento del volumen de datos relacionales.

Complementariamente, el proyecto demostró el impacto estructural que posee la indexación estratégica en tablas de alta densidad de registros. La creación de índices B-Tree específicos permitió reducir la latencia del percentil P99 en un 64%, validando que la optimización a nivel de persistencia ofrece un retorno de inversión superior a la lógica de aplicación en escenarios de saturación de recursos. Asimismo, la integración de Redis como capa de persistencia temporal resultó fundamental para desacoplar el flujo de autorización de la base de datos principal. Esta arquitectura de caché redujo la latencia extrema en un 68%, demostrando ser la solución técnica idónea para servicios con alta frecuencia de consulta y baja tasa de cambio en la información.

Finalmente, la formalización de estas mejoras a través de RFCs técnicos revisados por el equipo de ingeniería aseguró que las intervenciones cumplieran con los estándares de mantenibilidad y seguridad de la organización. Este proceso de documentación técnica garantiza que las optimizaciones sean sostenibles a largo plazo y proporciona una base sólida para la evolución de la infraestructura distribuida de la compañía. En conjunto, los resultados obtenidos confirman que la combinación de diagnósticos precisos, ajustes en la capa de persistencia y el uso de tecnologías de caché en memoria fortalece la estabilidad sistémica y prepara al backend para manejar ráfagas de tráfico masivo de manera predecible.

REFERENCIAS

[1] J. Dean and L. A. Barroso, "The Tail at Scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, Feb. 2013. [En línea]. Disponible: https://dl.acm.org/doi/10.1145/2408776.2408794

[2] B. H. Sigelman et al., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report, 2010. [En línea]. Disponible: https://research.google/pubs/pub36356/

[3] Y. Sambra, "Observability in Microservices: A Systematic Literature Review," ArXiv preprint arXiv:2209.00652, 2022. [En línea]. Disponible: https://arxiv.org/abs/2209.00652

[4] J. Silva, J. P. Correia, and L. Cruz, "Analyzing and detecting N+1 query problems in ORM-based applications," Journal of Systems and Software, vol. 125, pp. 24–40, 2017. [En línea]. Disponible: https://doi.org/10.1016/j.jss.2016.11.020

[5] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," Acta Informatica, vol. 1, no. 3, pp. 173–189, 1972. [En línea]. Disponible: https://link.springer.com/article/10.1007/BF00288683

[6] W. Wang et al., "Improving Performance of Key-Value Stores with Caching and Tiered Storage," IEEE Access, vol. 8, pp. 102341–102352, 2020. [En línea]. Disponible: https://ieeexplore.ieee.org/document/9099308

[7] Redis Ltd., “Redis,” [En línea]. Disponible: https://redis.io/

[8] Datadog, Inc., “Datadog Application Performance Monitoring (APM),” [En línea]. Disponible: https://www.datadoghq.com/

[9] Django Debug Toolbar, “Django Debug Toolbar,” [En línea]. Disponible: https://django-debug-toolbar.readthedocs.io/en/latest/

[10] Amazon Web Services, Inc., “Amazon Web Services (AWS),” [En línea]. Disponible: https://aws.amazon.com/es/

[11] Amazon Web Services, Inc., “Amazon Elastic Container Service (Amazon ECS),” [En línea]. Disponible: https://aws.amazon.com/es/ecs/

[12] The PostgreSQL Global Development Group, “PostgreSQL,” [En línea]. Disponible: https://www.postgresql.org/

[13] Eraser, “Eraser: AI co-pilot for technical design,” [En línea]. Disponible: https://www.eraser.io/

[14] Holistics Software, “dbdiagram.io: Database Relationship Diagrams Design Tool,” [En línea]. Disponible: https://dbdiagram.io

[15] Django Software Foundation, “Django Models,” [En línea]. Disponible: https://www.djangoproject.com/

[16] Carbon, “Carbon: Create and share beautiful images of your source code,” [En línea]. Disponible: https://carbon.now.sh/