

Análisis y diseño de una arquitectura de software para sistemas de georreferenciación de eventos urbanos basado en dispositivos móviles

Maximiliano Arnaldo Sepúlveda Alvear

maximiliano.sepulvea@usm.cl

Profesor guía: Dr. Marcello Visconti Zamora

Resumen: El desarrollo de proyectos de software conlleva múltiples desafíos, entre ellos, diseñar una arquitectura de software idónea para operar en entornos de producción real. Este trabajo analiza el caso de Geoviality, un sistema de georreferenciación de eventos urbanos que, debido a una implementación monolítica *ad hoc*, presentó alta deuda técnica, acoplamiento crítico y problemas de rendimiento. Para resolver esto, se diseñó y validó una propuesta de arquitectura distribuida, robusta y genérica, basada en microservicios y fundamentada por los principios del *Domain-Driven Design* (DDD). La propuesta aborda aspectos como gestión de eventos, procesamiento de imágenes y mejores estándares de seguridad para garantizar el cumplimiento normativo. La validación consistió en una prueba de concepto donde se desacopló el módulo de procesamiento en microservicios independientes. Los resultados demostraron una optimización técnica significativa: una reducción del 79% en el uso de almacenamiento, 83% en el uso de red y 99% en la latencia percibida por el usuario (tiempo de recepción) en escenarios de alta carga, a cambio de una leve sobrecarga en el tiempo de procesamiento. Finalmente, se verificó la viabilidad del escalado horizontal utilizando Kubernetes, logrando mitigar la presión en las colas de tareas y asegurando la alta disponibilidad del sistema.

Palabras Clave: arquitectura de software, microservicios, dispositivos móviles, sistemas de gestión geográfica, Domain-Driven Design

1 Introducción

1.1 Contexto del proyecto

Geoviality es un sistema de gestión geográfica enfocado en el análisis de eventos en la calzada (socavones, grietas, baches, deformaciones o desperfectos en el asfalto de las calles por donde transitan vehículos) utilizando inteligencia artificial y visión computacional. Desarrollado por un equipo de 6 estudiantes para la 32.^a Feria de Software de la Universidad Técnica Federico Santa María en 2024.

El objetivo de Geoviality es: ser una herramienta de automatización y centralización de información sobre el estado de las calzadas, para las municipalidades del país, ayudando a tener una vista macro del estado de las calles de la comuna. Con esto, se busca agilizar la toma de decisiones y la gestión oportuna de las reparaciones. Todo esto es con el fin de mejorar la calidad de vida y la seguridad de los transeúntes y conductores de la localidad. [1], [2]

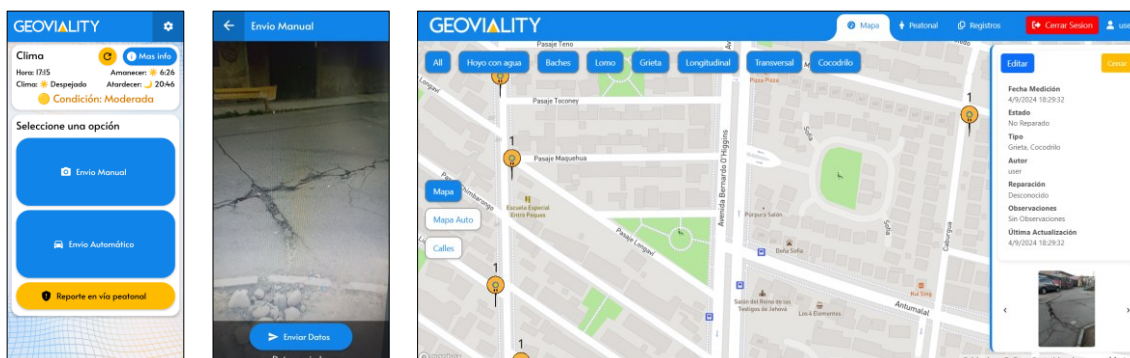


Figura 1. Interfaz móvil y web de Geoviality. Fuente: elaboración propia



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Análisis y diseño de una arquitectura de software para sistemas de georreferenciación de eventos urbanos basado en dispositivos móviles

Nombre del candidato(a): Maximiliano Arnaldo Sepúlveda Alvear

Carrera / Grado: Ingeniería Civil Informática

Campus: San Joaquín **Departamento:** Departamento de Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Marcello Alfredo Visconti Zamora, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

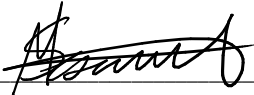
Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 25/02/2026

Firma: 

Estudiante o Candidato(a):

Fecha: 25/02/2026

Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

El levantamiento de datos se realiza con dispositivos móviles: utilizando GPS y capturas de imágenes de la calzada con la cámara. Todo lo anterior es automático mientras el usuario conduce o de manera manual si camina por las calles. Dichos datos son enviados por red a un servidor, en donde se procesan con un modelo de visión para clasificar y etiquetar el tipo de evento.

Una interfaz web sirve de vista para los datos levantados: se muestra un mapa con los puntos geolocalizados de los eventos detectados, junto con la imagen adjunta, fechas, estado, entre otra información relevante. El usuario puede registrar observaciones, realizar un seguimiento del estado de reparación, y ver analíticas históricas de los datos obtenidos.

1.2 Problema

El desarrollo del proyecto se realizó usando SCRUM con 3 *sprints* de 3 semanas, prototipado y recopilación de requerimientos. Sin embargo, considerando los cortos plazos de tiempo y por decisiones estratégicas del equipo, el proyecto careció de un diseño arquitectónico formal, resultando en una implementación *ad hoc*¹. En otras palabras, el desarrollo de código fue inmediato, se consideraron únicamente los requerimientos a corto plazo, y no se priorizaron atributos de calidad de software como la mantenibilidad y escalabilidad. Esto derivó en una implementación monolítica, caracterizada por su rigidez y alto acoplamiento, lo que generó una acumulación de deuda técnica significativa en etapas tempranas del proyecto.

- El desarrollo se veía constantemente bajo impedimentos: un cambio en el código causaba múltiples problemas no intencionados en otros aspectos del sistema, y que no necesariamente estaban relacionados.
- La organización interna del equipo fue incoherente: no existía un lenguaje único y estándar para referirse a los aspectos del negocio, por lo tanto, no había consistencia en la comunicación de ideas, ni en el código fuente. Además, no había documentación.
- Acoplamiento de procesos: la lógica del negocio estaba rígidamente acoplada en el código fuente, y no estaban parametrizadas. Múltiples módulos dependían fuertemente de otros, y el mínimo cambio en uno rompía la lógica de todo el resto.
- Problemas de rendimiento, como: comunicación síncrona en procesos lentos, uso redundante o excesivo de red y ausencia de políticas con respecto a la seguridad de los datos.

La combinación de todos estos problemas causó una reducción considerable en la eficiencia deseada de las metodologías ágiles. Además de una alta rigidez de desarrollo, interdependencia, y poca reutilización de código para otros fines dado el alto nivel de acoplamiento al negocio original. Si bien el sistema funcionaba, los estándares no eran adecuados para un entorno de producción real.

1.3 Propuesta y objetivos

En este trabajo, se analizará el estado actual del proyecto Geoviality bajo la visión de su arquitectura de software. El objetivo de este análisis es: identificar los puntos débiles, problemas latentes y malas prácticas realizadas; y luego proponer las bases conceptuales fundamentales y genéricas para sistemas similares a Geoviality. Es decir: sistemas que manejan georreferenciación de eventos urbanos en un sentido amplio, desde imperfectos en la calzada, eventos sociales, puntos de interés público, hasta problemas de seguridad como robos o accidentes.

Con dichas bases fundamentales, se busca que el diseño de este tipo de sistemas contemple una arquitectura de software más robusta y planificada desde el inicio. El objetivo es sistematizar el ciclo de desarrollo, alineándolo con los estándares de la industria y las metodologías de ingeniería, garantizando un desempeño óptimo en entornos de producción real. Además, considerando la sostenibilidad técnica a largo plazo, preparando al sistema para escenarios que requieran mayores estándares en atributos de calidad como la escalabilidad, la extensibilidad y la resiliencia.

En este trabajo, en particular, se propone una arquitectura de software basada en microservicios. El diseño arquitectónico se fundamenta en los principios del *Domain-Driven Design* (Diseño Guiado por el Dominio, DDD), estableciendo *Bounded Contexts* (Contextos Delimitados) claros para separar las múltiples

¹ Hecho específicamente para un fin preciso. En este caso, Feria de Software. No es generalizable para otros propósitos.

responsabilidades del negocio. Este paradigma asegura una alta cohesión interna y un bajo acoplamiento entre servicios, facilitando la implementación de estrategias de escalado para soportar múltiples usuarios y optimizar procesos. También se toman en consideración requerimientos no funcionales como la seguridad del sistema, la resiliencia y la disponibilidad.

1.4 Metodología de validación

La validación de esta propuesta se fundamenta en el desarrollo de una prueba de concepto aplicada sobre el proyecto Geoviality ya existente. Precisamente, se experimenta el desacople del módulo original de procesamiento en dos microservicios independientes: GeoData y Enrichment Worker, aislando los contextos de gestión de datos y procesamiento asíncrono.

El objetivo principal es realizar un análisis comparativo cuantitativo entre la arquitectura original y la propuesta basada en microservicios. Para ello, se evalúa el impacto de rendimiento y la eficiencia de los procesos, como el tiempo de recepción de datos (latencia percibida), el tiempo de cola de las tareas de procesamiento, el *back-pressure* de la cola de mensajes y el uso de recursos.

Con este experimento, se busca verificar la viabilidad técnica de separar contextos de negocio, y determinar cómo este cambio afecta tanto a la estructura del código, la comunicación entre servicios, y el rendimiento de los procesos de negocio. Asimismo, el análisis contempla la identificación de nuevos desafíos inherentes a los sistemas distribuidos, como el aumento de la complejidad operativa, los costos de mantenibilidad, el uso de recursos computacionales y estrategias de orquestación para garantizar escalabilidad.

1.5 Estructura del informe

En primer lugar, la actual introducción incluye un breve resumen del proyecto y su contexto, en donde se presenta la problemática planteada y la necesidad de una solución y su justificación.

En la sección 2, se presenta el marco teórico. Se da una introducción académica al lector sobre los conceptos que se van a abordar en el escrito. Además de algunas definiciones globales para establecer un lenguaje estándar en este informe.

En la sección 3, se realiza el análisis *as-is* de Geoviality. Se presenta la situación actual del proyecto, junto a su arquitectura y la estructura de sus procesos lógicos principales. Al final, se presenta un análisis de los problemas identificados y la necesidad de una solución.

La sección 4 consiste en la propuesta *to-be* de nueva arquitectura. Utilizando el análisis de la sección anterior, se presenta una planificación más general y extensa de diversos aspectos de la arquitectura de software del proyecto; tomando una vista más genérica y parametrizable. Aquí se implementa el *Domain-Driven Design* con los contextos y dominios del negocio, junto con algunos requerimientos no funcionales esenciales, la implementación de los microservicios, sus métodos de comunicación, y la especificación de algunos flujos lógicos principales.

La subsección 4.7 consiste en un análisis de los riesgos y desafíos que conlleva la implementación de esta arquitectura. Riesgos como la complejidad añadida de desarrollo, costos de mantenimiento y acoplamiento de servicios. Y desafíos como el rendimiento del sistema, la resiliencia, y por, sobre todo, el cumplimiento legislativo. Esto último se analiza en el contexto de la Ley 21.663 sobre Marco de Ciberseguridad, y especialmente la futura Ley 21.719 sobre Protección de Datos Personales.

La sección 5 presenta la metodología de validación. Se desarrolla una prueba de concepto (*proof of concept*) que implica el desacople parcial de uno de los módulos del proyecto original en microservicios. Comparando el rendimiento entre la implementación actual y la experimental utilizando métricas de calidad definidas. Además, se experimenta con la nueva capacidad de escalabilidad gracias a la nueva arquitectura.

Finalmente, la sección 6 sintetiza las conclusiones del análisis. Presentando los aprendizajes obtenidos, y así demostrando como una buena planificación previa de la arquitectura de software de un sistema puede ayudar a guiar el desarrollo de un proyecto, así como puede introducir nuevos desafíos y riesgos que es necesario considerar para mitigar a tiempo.

2 Marco teórico

Durante el desarrollo de este análisis, se mencionará múltiples conceptos y áreas de estudio. En esta sección se resumirán brevemente los conceptos más relevantes para enmarcar el contexto de conocimiento requerido y asegurar una comprensión óptima del escrito.

2.1 Arquitectura de software

La arquitectura de software consiste en un conjunto de decisiones fundamentales y de alto nivel sobre la estructura, comportamiento y evolución de un sistema informático. En otras palabras, toma en consideración toda la organización de un sistema informático, y guía decisiones conceptuales: las piezas que lo conforman, cómo se conforman, las relaciones entre las piezas, y el objetivo que el sistema como unidad busca lograr. Además, las arquitecturas son el vehículo principal para la consecución de los objetivos de negocio y los requisitos no funcionales, permitiendo realizar análisis previos o comparaciones frente a otras arquitecturas, incluso antes de la implementación práctica. [3], [4], [5]

Basar el desarrollo de un sistema bajo una arquitectura de software permite tener una vista más predecible del desarrollo y de su funcionamiento. Esto es una gran ayuda para los desarrolladores ya que, idealmente, no deberán crear soluciones desde cero, sino implementar las soluciones que ciertos patrones de diseño ya han demostrado funcionar. [6] No obstante, ninguna arquitectura es “correcta”. La implementación de una arquitectura en un proyecto de software implica un conjunto de compromisos o *trade-offs* que se deben gestionar y decidir activamente. [7]

2.1.1 Patrones de diseño

Si la arquitectura de software avala una vista más conceptual y de alto nivel sobre un sistema, un patrón de diseño es entonces la herramienta práctica para aterrizar la tesis de una arquitectura. La arquitectura es la estrategia estructural, y los patrones de diseño las tácticas operativas para resolver problemas recurrentes en la implementación.

En otras palabras: un patrón de diseño es una solución recurrente a un problema recurrente. En el caso de sistemas informáticos, un patrón de diseño es una solución probada² a un problema común que ocurre dentro de un contexto específico de una arquitectura de software. [8]

Los patrones de diseño se centran en aspectos más locales del desarrollo y la implementación práctica del sistema. No son librerías o extractos de código que se puedan introducir directamente, sino descripciones que buscan guiar o adaptar una implementación para resolver un problema. En cambio, una arquitectura de software avala un alcance más global y estratégico del sistema con alta abstracción, y establece decisiones que buscan cubrir ciertos aspectos no funcionales de un sistema.

2.1.2 Sistemas monolíticos

Un sistema monolítico consiste en un modelo de desarrollo de software más tradicional, en donde toda la lógica de negocio, los datos, y las interfaces están incluidos en una sola unidad atómica. Para efectos de un prototipo, una herramienta interna de alguna organización, o un producto que requiere distribuirse al público, un software monolítico puede ser una buena idea. Esto es porque los sistemas monolíticos son más baratos de implementar, más sencillos de desarrollar, y no requiere necesariamente una planificación extensa para comenzar a escribir código. Todo el paquete incluye lo necesario, toda la lógica está embebida, y no es divisible. [7]

Sin embargo, si los requerimientos del negocio cambian constantemente, se requiere escalar a muchos usuarios, o simplemente se necesita actualizar regularmente, un monolito puede volverse un problema a largo plazo ya que requiere cambiar y posiblemente recompilar todo el software. Y lo que alguna vez fue una

² Generalmente las soluciones son probadas de forma heurística, y han demostrado funcionar muy bien a través del tiempo. En otras palabras, son soluciones reutilizables y confiables a problemas de una misma naturaleza, porque se ha demostrado que funciona en la práctica durante mucho tiempo.

implementación rápida y barata en un inicio, se puede tornar en un sistema complejo y caro de mantener en los próximos años. Newman lo expresa como una “Gran Bola de Lodo” (*Big Ball of Mud*), queriendo decir: un monolito que crece constantemente eventualmente se degradará por el acoplamiento de sus componentes, terminando en fenómenos como el código espagueti. [9], [10]

2.1.3 Microservicios

A raíz de los problemas de agilidad y escalabilidad de los sistemas monolíticos, los microservicios emergieron como una alternativa. Consiste en un enfoque donde una aplicación se desarrolla como un conjunto de piezas llamados *servicios*: unidades de software muy pequeños, autónomos, y que trabajan en conjunto con otros servicios comunicándose a través de una red. Cada *servicio* funciona de forma independiente, con su propia lógica, y definido en un contexto específico de algún aspecto del negocio. [11]

La arquitectura de microservicios resulta bastante útil en casos que se requiera diseñar un sistema en torno a la estructura organizacional del negocio específico. Esto es: cada aspecto principal del negocio debería ser manejado de forma independiente y exclusiva por un sólo *servicio*. [9] Los patrones de diseño basado en microservicios han demostrado ser una muy buena solución a problemas sobre escalabilidad, disponibilidad y resiliencia de los sistemas. Con tecnologías como los contenedores se puede replicar un mismo servicio varias veces y en múltiples ubicaciones (tanto lógicas como geográficas), permitiendo manejar altas cargas y resistir fallas catastróficas sin afectar la disponibilidad de la aplicación hacia los clientes. [12], [13], [14]

A diferencia de sistemas monolíticos, los microservicios añaden una capa extra de complejidad, tanto en el desarrollo como en el despliegue y la mantenibilidad. Por lo tanto, se requiere entender y planificar más la estructura y las necesidades del negocio, para así poder aprovechar de forma correcta y eficiente los beneficios que entrega este tipo de arquitecturas. [10] En la actualidad, empresas del *Big Tech* como Netflix basan gran parte de su modelo de negocio en sistemas basados en microservicios. [15], [16]

2.1.4 Escalabilidad

La escalabilidad y la elasticidad consisten en la capacidad de un sistema para soportar una carga de trabajo que va creciendo constantemente, al mismo tiempo que se mantiene el rendimiento y la disponibilidad. Un sistema que espera crecer y servir a una multitud de usuarios debería considerar la escalabilidad como un atributo de calidad esencial. Existen dos estrategias para escalar un sistema: vertical y horizontal. [7]

El escalado vertical es el más tradicional. Para aumentar las capacidades de un sistema, se requiere añadir más recursos de hardware (CPU, RAM, almacenamiento) al servidor. Es la forma más sencilla de aumentar las capacidades de un sistema, sin embargo, es la más cara a largo plazo; eventualmente tiene un límite, ya sea físico o monetario. Además, no soluciona el problema del punto único de falla. Sigue siendo una única máquina, y si falla, todo el sistema cae.

El escalado horizontal es el estándar en la arquitectura de microservicios y computación en la nube. Aumentar las capacidades del sistema se traduce a: replicar el sistema o parte de sus componentes con el fin de repartir la carga de trabajo. En otras palabras, en vez de mejorar las características del servidor, se añade una o más “copias” de ese mismo servidor y se introduce un mecanismo para balancear la carga de trabajo entre esas máquinas. Esto, además de ser capaz de soportar grandes cargas, añade una capa de resiliencia al sistema: si uno de los servidores falla, no falla todo el sistema, ya que el resto de los servidores siguen operativos y pueden ser usados como reemplazo de forma temporal. [17]

2.1.5 Estilos de comunicación

En el contexto de sistemas distribuidos, la comunicación es un aspecto esencial para relacionar los componentes de un sistema; ya que no se considera únicamente llamadas internas dentro de un mismo proceso, sino que también llamadas a otros procesos; incluso a procesos de otras máquinas a través de la red. Esto último añade una capa de complejidad que requiere tomar decisiones fundamentales sobre el flujo lógico de los procesos y las comunicaciones; por ejemplo: la latencia, el manejo y sincronización de los datos entre sistemas y el acoplamiento lógico. Existen dos estilos principales: síncrono y asíncrono. [18], [19]

La comunicación síncrona se refiere cuando un proceso llama a otro proceso y su operación se “bloquea” hasta recibir la respuesta del otro proceso. Esto es útil cuando necesariamente se necesita la información del otro proceso para continuar con la operación. Un patrón que se basa en este estilo es *request-response*, es decir, un proceso solicita algo y luego espera a una respuesta antes de continuar. Si algo falla, la consulta se reintenta. Protocolos como REST sobre HTTP o gRPC utilizan este estilo de comunicación. Es simple de implementar, útil para operaciones de autenticación, lecturas de datos rápidos u operaciones que requieren inmediatez, sin embargo, por su naturaleza bloqueante, produce acoplamiento temporal. En otras palabras, el rendimiento o éxito de la operación depende de la latencia de respuesta (incluyendo cadenas de consultas), o de la disponibilidad del otro proceso respectivamente (si el servidor se cae, el cliente se bloquea).

La comunicación asíncrona se refiere cuando un proceso, al llamar a otro proceso, no se “bloquea”; es decir, se puede continuar con la operación sin recibir la respuesta aún. O también, al llamar a otro proceso, ni siquiera se espera respuesta, simplemente se asume que “eventualmente” llegarán los datos al otro proceso. Una arquitectura que se basa en este estilo de comunicación es *event-driven*, es decir, los procesos están a la escucha de alguna notificación que les indique comenzar una tarea, o, por otro lado, un proceso emite una notificación, pero a diferencia de *request-response*, no espera una respuesta. Protocolos como RabbitMQ, Kafka, o Webhooks soportan este estilo de comunicación. Gracias a su naturaleza no bloqueante, permite que un programa pueda hacer otras tareas después de hacer una consulta o emitir un mensaje. Útil en situaciones de tareas de larga duración, o comunicar eventos a diversos microservicios que no requieren tomar acciones inmediatas. Esto permite desacoplar el proceso de otros, permitiendo que funcione de manera independiente. Al mismo tiempo, mejora la resiliencia ya que, si un proceso no responde, no bloquea todo el flujo.

No obstante, la complejidad recae en solucionar la consistencia de los datos entre procesos, ya que dos procesos que quieren acceder al mismo dato podrían recibir datos diferentes, desactualizados, o repetidos. Uno de los modelos de consistencia clásicos en arquitecturas distribuidas, y la que se verá en esta propuesta, es *read-your-writes* (lee tus escrituras). Cuando un proceso actualiza un dato, cualquier intento posterior de lectura del mismo proceso siempre devolverá el valor actualizado, y no uno obsoleto. [20] Esto es importante no solo para la integración y coherencia entre los microservicios, también para la experiencia de usuario.

2.2 Tecnologías y herramientas

El uso de tecnologías estándar permite que un proyecto pueda estar más alineado con las expectativas de funcionamiento en un entorno real. Por esta razón, no es ideal crear soluciones de software desde cero sin antes investigar si ya existe una tecnología o herramienta que permita solucionar un problema. Normalmente, tecnologías estándar ya están probadas, son mantenidas regularmente, y tienen un alto grado de confiabilidad y seguridad; a diferencia de implementaciones propias que podrían no estar considerando alguno de esos aspectos. [21]

Para efectos de esta propuesta, se busca evitar la duplicidad de esfuerzos de desarrollo. En otras palabras, se busca integrar tecnologías, herramientas o frameworks que ya existen, estén probadas, y sean un estándar en la industria.

2.2.1 Sistemas de Información Geográfica (SIG)

Un Sistema de Información Geográfica o SIG (*Geographic Information System* o GIS) es un sistema de software diseñado para capturar, almacenar, manipular, analizar, administrar y presentar datos geográficos o referencias espaciales. No necesariamente incluye mapas, sino que conlleva una colección completa de herramientas y procesos internos que permiten obtener inteligencia adicional de los datos geográficos que uno posee. [22], [23]

Asimismo, internamente los SIG necesitan manipular datos en formatos específicos y estándar para permitir interoperabilidad entre sistemas. Los formatos más relevantes para este contexto son:

- GeoJSON: un formato basado en JSON para codificar estructuras de datos geográficos. Es ampliamente utilizado para mostrar información en mapas, o para representar geometrías. GeoJSON es un formato altamente portable, permitiendo compatibilidad con diferentes sistemas. [24]

- Teselas Vectoriales (Vector Map Tiles): a diferencia de las teselas rasterizadas, que corresponden a imágenes estáticas, las teselas vectoriales son una representación más geométrica de las características de una sección de un mapa³. Permiten almacenar un mapa con alto detalle, pero usando menos almacenamiento. Mapbox definió una especificación abierta sobre como codificar y utilizar teselas vectoriales en la visualización de mapas. A raíz de esto, múltiples librerías e implementaciones han salido a la luz. [25]
- PostGIS: es una extensión de PostgreSQL optimizado para almacenar datos geoespaciales. Es el software de base de datos estándar para almacenar este tipo de información. Incluye índices y consultas optimizadas para realizar búsquedas geométricas (distancias, curvas, caminos). [26]

Geoviality se inspira fuertemente en los SIG, ya que: a partir de información inicial como una imagen georreferenciada, el sistema era capaz de analizar automáticamente dichos datos, y ofrecer nuevos datos. La agregación de datos a lo largo del tiempo también permite al sistema crear nuevo conocimiento. Además, se ofrece al usuario una forma centralizada y simple de visualizar los datos, y administrarlos.

Como todo sistema de información, es esencial obtener los datos en primer lugar. Geoviality hace práctica de la Información Geográfica Voluntaria (Volunteered Geographic Information, VGI). Consiste en la producción, recopilación y difusión de datos geográficos por parte de ciudadanos o individuos particulares utilizando herramientas (en este caso, un dispositivo móvil) para dicha finalidad, con menos costos asociados. Sistemas de este estilo (por ejemplo, Waze o SSAFE) deben confiar en la cooperación colectiva para levantar información. El SIG se encarga de recopilar, procesar y asegurar la calidad de los datos. [27]

2.2.2 Contenedores y orquestación

Los contenedores y plataformas de orquestación de contenedores son relevantes ya que estos componentes de software permitieron que la arquitectura de microservicios sea viable técnicamente. Y en la actualidad, son tecnologías estándar para sistemas basados en microservicios.

Un contenedor es un componente de software que permite empaquetar código y dependencias de un programa informático en una unidad aislada y replicable. Permite desplegar una aplicación garantizando que funcione exactamente igual en diferentes máquinas. Toma conceptos similares a los de una máquina virtual, pero es más ligero en su ejecución. Cada componente de un sistema se ejecuta en su propio contenedor, comunicándose con otros a través de red. Docker es una de las múltiples plataformas de contenedores que existen y uno de los más populares. Cuando se busca escalar un sistema desplegado en contenedores en múltiples máquinas físicas, y a buscar una administración más avanzada de estos, entran en juego los orquestadores. [28]

Un orquestador de contenedores es una plataforma de software que permite administrar el ciclo de vida de múltiples contenedores, incluso a través de diferentes máquinas físicas conectadas por red en lo que se define como un clúster. Se define un “estado deseado” y el orquestador automáticamente se encarga de mantenerlo: creando o eliminando contenedores, creando replicas, estableciendo conexiones de red, balanceando cargas, y más. Kubernetes es una de las múltiples soluciones de orquestación que existen, y el estándar en la industria, utilizado por múltiples proveedores de *cloud*. [14], [29]

2.3 Seguridad de la información

La seguridad de la información es una disciplina en la informática dedicada a proteger los activos de información contra el acceso, uso, divulgación, interrupción, modificación, o destrucción no autorizada. En esta área, se investigan metodologías y herramientas con el objetivo de garantizar que la información sea accesible y confiable para quienes la necesiten.

Los 3 pilares de la seguridad de la información se conocen como la *Triada CIA*, cuyos pilares son: [30], [31]

- Confidencialidad: garantizar que la información sea accesible únicamente para aquellas entidades que están permitidas verlas. Conceptos como el cifrado y la comunicación privada soportan este pilar.

³ Análogo a imágenes digitales: una imagen rasterizada (píxeles) puede ser un PNG, una imagen vectorial (curvas matemáticas) puede ser un SVG.

- **Integridad:** garantizar que la información sea original, precisa, y completa, sin alteraciones no autorizadas. Conceptos como el hashing y las firmas digitales soportan este pilar.
- **Disponibilidad (*Availability*):** garantizar que los sistemas y los datos estén siempre funcionales y disponibles cuando se requiera por usuarios autorizados. Un ejemplo que soporta este pilar es la arquitectura de microservicios, y sus patrones de diseño basado en escalado, replicas, y resiliencia.

La seguridad es un proceso continuo y no un producto. Si bien las técnicas de criptografía son necesarias, no son suficientes; se debe asumir que los componentes de un sistema fallarán y serán comprometidos. [32] Para mantener seguridad en un sistema se debe tener control sobre quienes tienen acceso y cómo. Para ello, hay que aplicar procesos y mecanismos de autenticación y autorización. Estos dos conceptos están muy relacionados, incluso podrían confundirse, sin embargo, avalan aspectos diferentes en el mecanismo de control de acceso a un sistema.

La autenticación se refiere a un proceso de verificación de identidad. Busca responder la pregunta “¿Quién eres?”. Por ejemplo, sistemas donde se requiere un inicio de sesión. La autorización, en cambio, se refiere a un proceso de definición de permisos. Busca responder la pregunta “¿Qué puedes hacer?”. Un ejemplo sencillo, son los administradores de base de datos en donde se pueden definir roles o permisos a los usuarios. Los roles definen a que contenido tienen acceso y que tipo de acciones puede realizar.

2.3.1 JSON Web Tokens (JWT)

Uno de los mecanismos prácticos para aplicar procesos de autorización y autenticación en un sistema, es a través del uso de JSON Web Tokens. Un JSON Web Token (JWT) es una estructura de datos en formato JSON, pero está codificado y firmado criptográficamente. Corresponde a un estándar muy común para la transmisión segura de información entre pares, que busca ser compacto y autocontenido.

Técnicamente está conformado por un *header* (información sobre el algoritmo de la firma), un *payload* (información **no sensible** para identificar a un usuario) y un *signature* (la firma criptográfica, asegura integridad y autenticidad). Tanto el *header* como el *payload* tiene campos clave-valor llamados *claims*. Emitir un JWT es análogo a “prestar una llave” que posee una etiqueta con información del usuario autenticado. Presentar esa llave frente a los sistemas nos permitirá obtener acceso a la información protegida. Como dicha llave posee una firma criptográfica, se puede verificar y garantizar que no fue alterada por un tercero. Por lo tanto, no se pueden falsificar, sin embargo, no es inmune a vulnerabilidades.

Es importante recalcar los detalles de seguridad de los JWT, ya que por sí solos no introducen ningún mecanismo de seguridad al sistema que los utiliza. En primer lugar, se asume que todo lo que entra en el *payload* es información pública. Un JWT no debe incluir indicios de contraseña ni cualquier otra información clasificada del usuario; además es un objeto transferible y no caduca por defecto. Un JWT funciona bajo el principio de “quien lo posee, lo usa”, esto significa que el token de un usuario puede ser copiado o transferido a otro ente, y este puede usarlo para acceder en nombre del usuario original para siempre. Esto se conoce como un ataque de *token replay* (una forma de *spoofing* o suplantación). [33]

La seguridad de un JWT recae en el hecho de que es un objeto inalterable. Un sistema que verifica exitosamente un JWT solo puede concluir que el token y su información es legítimo y auténtico, pero no necesariamente que el ente quien lo presenta es realmente el dueño original del token. Para que un JWT realmente pueda ser usado como mecanismo de seguridad, es importante establecer políticas de expiración, proteger la clave de cifrado, o usar algoritmos de firma más robustas como algoritmos asimétricos con pares de clave pública/privada. Ejemplos de esto puede ser la separación de *Access Tokens* y *Refresh Tokens* con algoritmos asimétricos como RS256.

2.3.2 Demonstrating Proof-of-Possession (DPoP)

La Demostración de Prueba de Posesión es un mecanismo de seguridad que busca mitigar ataques de suplantación mediante la vinculación criptográfica del *Access Token* a una instancia de cliente específica. El protocolo exige que el cliente demuestre la tenencia de su clave privada (firmando un token de prueba) a través de HTTP. De esta forma, el servidor puede validar que el cliente quien presenta el *Access Token* es realmente quien posee la clave privada asociada a este, previniendo ataques de suplantación como *token replay*.

- Cada dispositivo cliente genera un par de claves pública y privada. Al momento de solicitar un *Access Token*, la consulta HTTP incluye un *header DPoP*. Este *header* corresponde a una “demostración de posesión”: es un JSON Web Key (JWK)⁴ que incluye la clave pública del dispositivo, información sobre la consulta HTTP, y está firmado con la clave privada.
- El servidor recibe la consulta HTTP con la “demostración de posesión”, extrae la clave pública y verifica la firma del JWK. Si la verificación es exitosa, se garantiza que la clave pública no fue alterada, y con esto, el autor del JWK demostró que tiene posesión a la clave privada del dispositivo.
- El servidor vincula el JWK del dispositivo al nuevo *Access Token*: en su *payload* se añade un campo nuevo llamado “confirmación” y allí se registra el JSON Key Thumbprint (JKT) del JWK. En otras palabras, en lugar de guardar la clave pública directamente, se guarda el hash de este. Finalmente, el *Access Token* “vinculado” se envía al cliente.
- Para posteriores consultas en donde se requiera autorizar el dispositivo, el cliente deberá enviar su *Access Token* “vinculado”, junto a una nueva prueba DPoP en el header.

Este mecanismo permite que un servidor pueda verificar: 1) La integridad del *Access Token*: es decir, fue emitido por un sistema autorizado y es legítimo. 2) La posesión del dispositivo: es decir, el autor del JWK tiene acceso a la clave privada del dispositivo. 3) La vinculación del *Access Token* con el dispositivo: es decir, la clave pública incluida en la prueba DPoP coincide con el hash que está presente en el *Access Token* “vinculado”. [34]

Si esas 3 verificaciones coinciden, entonces se garantiza que el *Access Token* que el cliente presentó efectivamente está vinculado al dispositivo y, además, se tiene acceso a su clave privada; es decir, se demuestra la posesión del dispositivo. Véase el Anexo D y el Anexo E, que incluyen un diagrama del funcionamiento con pseudocódigo de implementación.

3 Diagnóstico As-Is

El presente capítulo expone un análisis técnico sobre el estado original del proyecto Geoviality. El objetivo de este análisis es presentar las limitaciones estructurales y operativas a causa de la implementación actual.

3.1 Arquitectura

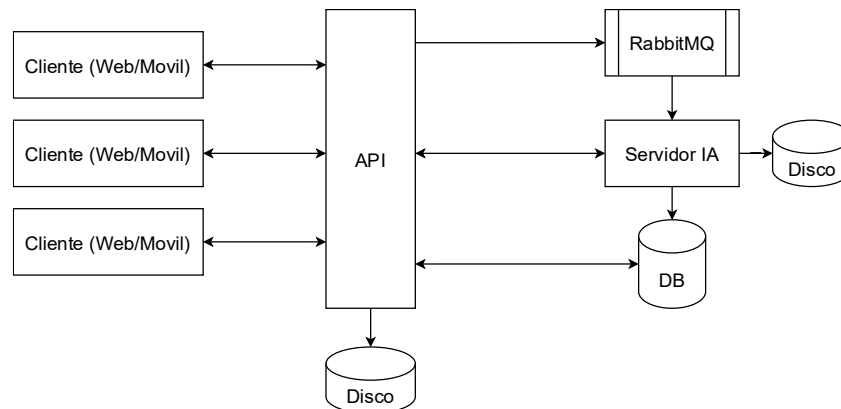


Figura 2. Arquitectura original de Geoviality. Fuente: elaboración propia

Durante el desarrollo se intentó seguir un patrón de microservicios, pero el resultado fue más un patrón monolítico por capas, similar a Modelo-Vista-Controlador. En este caso, el componente controlador del sistema es la API, quien gestiona la autenticación y el traslado de información. Por otro lado, el servidor IA tomó el rol del modelo: un proceso separado de la API quien tiene la tarea de posprocesar los archivos utilizando YOLOv8⁵ y guardar la información en la base de datos según las reglas de negocio.

⁴ JWT y JWK son prácticamente lo mismo en términos de formato, pero se diferencian por su propósito. JWT se utiliza para autenticar y autorizar usuarios. Mientras que JWK se utiliza para transportar claves de cifrado (que se usan para verificar un JWT) en forma segura.

⁵ Algoritmo de detección de objetos por visión computacional. Versión 8 lanzada el 10 de enero de 2023. <https://docs.ultralytics.com/models/yolov8/>

La comunicación entre los clientes y la API es REST sobre HTTP. Por otro lado, entre la API y el servidor IA se realiza a través de colas de mensajería y REST al mismo tiempo. Además de una base de datos única, tanto la API como el servidor IA almacenan sus datos en discos propios. Véase la Figura 2.

3.2 Procesos principales

3.2.1 Registro de eventos y procesamiento

- 1) El proceso comienza cuando un cliente en móvil captura una imagen. Este es enviado hacia la API junto con metadatos asociados. La API al recibir los datos, genera y guarda una **ID de imagen** única y **envía todo a través de cola RabbitMQ**: el mensaje contiene los metadatos formateados en formato JSON y la imagen codificada en Latin-1 (ISO 8859-1). La imagen aún no es guardada en disco.
- 2) El proceso IA consulta cada 5 segundos para verificar si hay nuevos mensajes en la cola RabbitMQ. De ser así, se extrae los metadatos y se decodifica la imagen. Esta imagen **se guarda temporalmente** en su disco (diferente al disco de la API) y ejecuta el proceso de análisis con YOLO sobre esta, generando una **imagen anotada** y un **resultado** numérico.
 - a) Si el algoritmo detecta un evento, entonces la **imagen anotada** se mueve a otra ubicación y el **resultado** se adjunta en un archivo CSV (incluyendo información como ID, clase de objeto y confianza).
 - b) Si el algoritmo no detecta nada, entonces la **imagen anotada** se borra y el **resultado** se descarta.
- 3) Si se detectó un evento, entonces se procede con el guardado en la base de datos. Con las coordenadas de la imagen:
 - a) Se busca un **evento cercano** (<10 [m]) que ya exista en la base de datos. Si no existe, entonces se genera un nuevo registro con un **ID de evento** único y se adjunta la **ID de la imagen** allí. En caso contrario, se adjunta la **ID de la imagen** al **ID de evento** ya existente.⁶
 - b) Se busca la **calle más cercana** y se le incrementa un contador por tipo de irregularidad detectada.
- 4) Si se detectó un evento, entonces el proceso IA envía la **imagen original** (sin anotar) de vuelta a la API **a través de REST**. La API obtiene la imagen y finalmente lo guarda permanentemente en su propio disco. Al mismo tiempo, el proceso IA elimina la imagen temporal de su disco. Cuando un cliente solicita una imagen, la API es quien las carga y las envía.

Notar que la gran cantidad de manipulación de base de datos ocurre en el proceso IA. Además, los datos del evento que los clientes envían no se guardan hasta que todo el proceso de análisis con IA haya terminado antes. Un diagrama de secuencia de esta descripción se encuentra en el Anexo A.

3.2.2 Autenticación y autorización

El inicio de sesión se realiza generando un JWT para autenticar a los usuarios. Este token incluye únicamente el nombre de usuario y una fecha de expiración (8 horas). El token se firma utilizando el algoritmo simétrico HS256 con una clave secreta definida en variables de entorno. El cliente lo utiliza como *Bearer Token*⁷ en las siguientes consultas que se requieran. Se verifica la firma y decodifica payload, luego se verifica la fecha de expiración, el nombre de usuario y que dicho usuario no esté deshabilitado.

Los usuarios se guardan en la base de datos. La API verifica constantemente si el usuario está habilitado. La eliminación de usuarios es no destructiva, es decir, se marcan como “deshabilitados” en vez de eliminarse completamente. Esto ayuda a evitar pérdida de información.

Originalmente se quiso implementar una separación de permisos de usuario a través de 3 roles: Usuario, Analista y Administrador. Esto con el fin de mejorar el control sobre lo que los usuarios pueden hacer en el sistema (véase la Tabla 1). Sin embargo, esto solo fue implementado para permitir acceso a CRUD de administración de usuarios. También se quiso implementar un sistema de permisos de dispositivo. Esto con el fin de evitar que un cliente pudiese capturar imágenes desde un dispositivo no autorizado o evitar el uso de múltiples dispositivos sin autorización previa de la administración. Finalmente, no se implementó.

⁶ El ID de evento y el ID de imagen son identificadores diferentes. Puede haber múltiples imágenes asociadas a un mismo evento.

⁷ Un tipo de credencial incluido en el header HTTP “Authorization”. Es usado para autorizar acceso a recursos protegidos por una API.

Tabla 1. Roles de permisos planificados.

Acción/Rol	Usuario	Analista	Administrador
Iniciar sesión en cliente móvil	✓	✓	✓
Capturar imágenes	✓	✓	✓
Iniciar sesión en cliente web	✓	✓	✓
Generar reportes		✓	✓
Administrar eventos		✓	✓
Administrar usuarios			✓

3.2.3 Notificaciones

Cuando IA consigue una detección positiva, se guarda un registro en memoria: una cola para enviar notificaciones a móvil (notificar que la captura fue exitosa) y otra cola para enviar notificaciones a la interfaz web (alertar sobre nuevo punto y mover el mapa automáticamente).

Ambas colas son consumidas por los clientes móvil y web a través de dos endpoints que el servidor IA expone. Estas rutas contienen un *streaming* de datos *push* usando *Server-Sent Events* (SSE) a través de HTTP/1.1 sin autenticación. Al llamar la ruta, el cliente mantiene una conexión persistente con el servidor y puede recibir datos en vivo.

3.3 Identificación de problemas

Con el análisis anterior, se realizan observaciones respecto a problemas evidentes en la implementación actual. Para este caso, los problemas identificados se clasifican en las siguientes categorías:

- **Redundancia:** se refiere a un proceso que se repite innecesariamente. Principalmente un problema de optimización, que puede ser solucionado cambiando la lógica del código.
- **Inconsistencia:** se refiere a un proceso que tiene una lógica definida, pero luego, el mismo proceso se realiza de otra forma. También se refiere a cuando un proceso realiza una tarea sin seguir estándares fijos.
- **Acoplamiento:** se refiere cuando un proceso depende de otro proceso para realizar su tarea. También se refiere a un “conflicto de interés”: cuando un componente realiza una tarea que debería realizar otro.
- **Seguridad:** se refiere a problemas que puedan comprometer la seguridad de los datos internos del sistema.
- **Impedimento:** se refiere a una limitante técnica de una implementación. Solucionar el problema de raíz requiere una reimplementación o el uso de otra tecnología o solución.

Tabla 2. Problemas identificados en el registro de eventos y procesamiento

Clasificación	Descripción	Consecuencia
Redundancia	API recibe la misma imagen que envió hacia el servidor IA.	Duplicación innecesaria del uso de red.
Inconsistencia	API envía imagen hacia IA por cola asíncrona, pero IA envía imagen hacia API por REST síncrono.	Potencial DDoS. Envío de archivos síncrono podría bloquear IA hasta que termine.
Acoplamiento	IA realiza modificaciones a la base de datos.	Las reglas de negocio se encuentran acopladas a un componente inestable.
Acoplamiento	El proceso de registro de eventos depende del servidor IA.	Si el servidor IA se bloquea, las reglas de negocio también. La base de datos no podrá actualizarse ni recuperarse.
Redundancia	IA guarda copias de imágenes en disco, que luego son eliminadas.	Potencial degradación de disco, afectando al rendimiento a largo plazo.
Acoplamiento	API no guarda ni obtiene la imagen nueva hasta que la IA termine.	Pérdida de disponibilidad con información inmediatamente ingresada. No se cumple consistencia <i>read-your-writes</i> .
Inconsistencia	API envía imagen codificada en Latin-1.	Latin-1 no es estándar para enviar imágenes. Corrupción si la codificación no se maneja bien.

Tabla 3. Problemas identificados en la autenticación y autorización

Clasificación	Descripción	Consecuencia
Seguridad	Las sesiones validas solo dependen de la fecha de expiración.	No hay control granular ni arbitraria sobre las sesiones de usuario. No se pueden revocar en caso de ser necesario.
Inconsistencia	No hay implementado Refresh Tokens.	Las sesiones se cerrarán después de un periodo de tiempo incondicionalmente, afectando potencialmente a la experiencia de usuario.
Inconsistencia	La fecha de expiración depende exclusivamente de la zona horaria chilena.	No hay generalización del sistema para otros países. Cambios a horarios de verano podrían afectar los registros.
Inconsistencia	No se hace uso de campos estándar de JWT como "iat", "iss" o "aud". [35]	Integración con otras tecnologías o servicios web potencialmente afectada. Por ejemplo, a través de OAuth.

Tabla 4. Problemas identificados en las notificaciones

Clasificación	Descripción	Consecuencia
Impedimento	SSE no es escalable por defecto. El servidor tiene que mantener una lista de clientes conectados en memoria.	Para escalar se requiere una configuración específica, y asignar recursos del servidor.
Seguridad	Las notificaciones no están autenticadas.	Cualquier ente podría acceder y obtener datos potencialmente privados (coordenadas).
Redundancia	Las notificaciones se acumulan en memoria mientras no haya clientes que los consuman.	Potencial <i>back-pressure</i> en las colas, ocupando memoria innecesariamente.

3.4 Objetivos de nueva arquitectura

Si se plantea re-implementar este sistema, teniendo en cuenta los errores anteriores, entonces se debe tener en consideración los siguientes objetivos:

- La seguridad de los datos debe ser considerada desde el inicio de la planificación.
 - Sistemas de este estilo trabaja constantemente con información sensible, como lo son coordenadas físicas, archivos o fotografías.
 - Tener en cuenta la seguridad de los datos desde la planificación del sistema permite que su implementación sea menos problemática.
 - Dependiendo del país en donde se implemente, es necesario cumplir con requerimientos legislativos. En el caso de Chile: la Ley 21.663 sobre Marco de Ciberseguridad [36] y la futura Ley 21.719 sobre Protección de Datos personales. [37]
- Los módulos que componen el sistema deben dedicarse a un único aspecto del negocio.
 - Esto busca evitar el problema de conflictos de intereses entre los componentes del sistema. Por ejemplo: el servidor IA no debería gestionar la base de datos.
 - Para implementar esto, se puede explorar: 1) modificar el monolito e intentar desacoplar la lógica de negocio o 2) implementar y migrar a una arquitectura de microservicios.
 - Seguir la filosofía de la arquitectura de microservicios implicaría reducir el marco de acción de diversos servicios a lo más simple.
- Optimizar procesos largos aprovechando comunicación asíncrona.
 - Aprovechar una correcta implementación de comunicación asíncrona permitiría mantener el sistema disponible sin que un proceso ralentice todo. Y desde el punto de vista del usuario, puede mejorar la consistencia read-your-writes y la capacidad de respuesta (*responsiveness*).
 - En este caso, el servidor IA representa el mayor cuello de botella con respecto a los tiempos de procesamiento. Una buena forma de optimizar los tiempos puede ser: poner a disposición metadatos e imágenes inmediatamente después de recibirlas, mientras que el servidor IA sigue procesándolos.

Para esta propuesta, se seguirá con la estrategia de modelar una arquitectura basada en microservicios.

4 Propuesta To-Be

Como propuesta de nueva arquitectura, se considera un contexto genérico de Geoviality. Los 3 conceptos principales de este sistema son: 1) referenciación geográfica, 2) procesamiento/detección, y 3) dispositivos móviles. Con esto, el objetivo del proyecto se redefine a: “georreferenciar eventos urbanos utilizando dispositivos móviles como herramienta de recolección, con capacidad opcional de posprocesamiento.”

Domain-Driven Design (DDD) es un concepto que propone representar de mejor manera las fronteras de funcionalidad de los módulos de un sistema, basándose en los contextos en donde estos deben desenvolverse. [38], [39] En el contexto de arquitecturas distribuidas, este análisis es esencial para definir de forma concreta las tareas que cada servicio debe realizar. En otras palabras, lograr construir un sistema en donde sus componentes tienen un bajo acoplamiento y alta cohesión.

4.1 Lenguaje ubicuo

Uno de los pilares fundamentales del DDD es definir y adoptar un lenguaje único y estándar para el contexto de este proyecto. Hacer esto permite ahorrar tiempo y disminuir la “fricción de conceptos” entre los desarrolladores y los expertos del negocio. Todo esto, con tal de que ambos equipos hablen “el mismo idioma” y tengan una visión común. Para el contexto propuesto, se identifican los siguientes términos:

- **Evento:** fenómeno físico o social que ocurre en algún lugar de una ciudad o zona. El término es genérico y puede asociarse a diferentes contextos: desperfectos en la calzada, problemas de seguridad (robos, accidentes), eventos sociales, puntos de interés, entre otros.
- **Metadato:** información adicional en texto plano asociado a un **evento**. Es agregado manualmente por el usuario o automáticamente por el propio sistema o un dispositivo con sensores. Por ejemplo: coordenadas GPS, fecha de captura, tipo de evento, autor, observaciones, estado, y más de ser necesario.
- **Archivo:** un archivo digital asociado a un **evento**. Ejemplos: imágenes del lugar, archivos de evidencias, documentos de una inspección, y más del estilo.
- **Cliente:** programa informático que actúa como puerta de entrada para interactuar con el sistema. Puede ser una interfaz web en un navegador, como una aplicación móvil.
- **Usuario:** persona que interactúa con el sistema a través de un **cliente**. Tanto los **eventos** registrados, como sus **metadatos** y **archivos** quedan asociados a un usuario como “autor”.
- **Dispositivo:** pieza de hardware que tiene instalado un **cliente**, con el cual un **usuario** lo utiliza para capturar un **evento**. En este contexto, se refiere al dispositivo móvil.
- **Enriquecedor:** herramienta de software utilizada para posprocesar **archivos** y/o **metadatos**. El objetivo es generar y añadir más valor a la información recolectada. El término es genérico, y se relaciona con la necesidad de análisis. Ejemplos: un modelo de IA, clasificación de imágenes, detección por visión computacional, algoritmos de estadística o analítica o procesos de análisis histórico.
- **Resultado:** información cuantitativa generada por el **enriquecedor**. Se adjunta al **evento** en forma de **metadato** o **archivo**. Ejemplo: números, pesos, anotaciones en imágenes, rangos, etc.
- **Mapa:** vista gráfica de las ubicaciones de los **eventos**. Se accede a través del **cliente** web y es el principal método para ver y modificar los **metadatos** o los **archivos** de los **eventos**.

4.2 Dominios

Con los conceptos nombrados anteriormente, se puede definir la forma en cómo estos se relacionan entre sí; esto se expresa en la Figura 3. Con lo anterior, se puede identificar los siguientes dominios fundamentales en los que el sistema se basa:

- **Identidad y Acceso:** autenticación, autorización y gestión de sesiones. Los usuarios están asociados a roles con permisos definidos que limitan el acceso a ciertas partes o funciones del sistema. Por otro lado, también considera la gestión de dispositivos autorizados para acceder al sistema.
- **Gestión de eventos:** gestión y almacenamiento de eventos y sus metadatos asociados. Se considera como fuente de verdad, así, otros dominios pueden consultar. Incluye la integridad y la lógica de negocio, como agrupar eventos cercanos, evitar duplicados, generar informes, inteligencia de negocio, entre otros.

- **Almacenamiento de medios:** almacén de archivos binarios. Esto incluye datos anexos a los eventos, pero más pesados que los metadatos; como imágenes o documentos. Gestiona el ciclo de vida de los archivos, incluyendo el acceso, la carga, y la asociación archivo-evento.
- **Procesamiento de datos:** mecanismos de posprocesamiento de información. El Enriquecedor es el componente principal y el encargado de digerir datos de interés, quien luego genera inteligencia adicional y así, agrega valor a los datos del sistema. Este dominio debe manejar su propia lógica y no mutar datos externos. Gracias a esta naturaleza independiente, y según el contexto de negocio, este dominio puede considerarse opcional sin afectar el núcleo de la arquitectura ni la funcionalidad principal del sistema.
- **Visualización de mapas:** se puede considerar como una interfaz que tiene la tarea de recolectar los datos del sistema (agregación) y presentarlos en formato de “informe” que, en este caso, sería para mapas, por ejemplo, GeoJSON. A diferencia del dominio de gestión de eventos, este dominio se encarga principalmente de lectura y presentación de datos hacia el usuario.

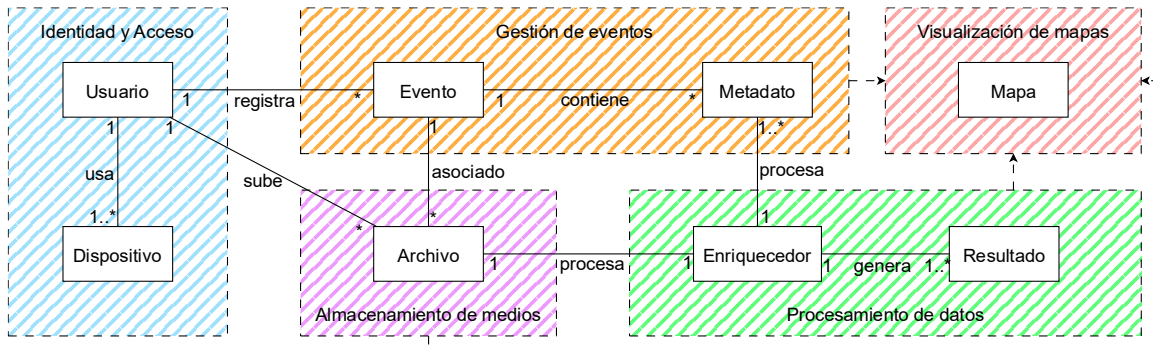


Figura 3. Dominios del sistema. Fuente: elaboración propia

4.3 Requerimientos no funcionales

Considerando los errores cometidos en la antigua implementación, se destilan los siguientes requerimientos no funcionales mínimos que un sistema de esta naturaleza debería tener en consideración.

4.3.1 Seguridad

La seguridad es crucial y debería ser uno de los pilares más fundamentales. Hay que recordar que en este tipo de sistemas se trabaja con ubicaciones, coordenadas, y archivos asociados a usuarios específicos. No darle importancia abre paso a que, frente a incidentes de seguridad, puedan ocurrir robos de credenciales y filtrado de datos que pueden ser explotados por actores maliciosos. Principalmente en Chile, se deben aplicar políticas e implementar requerimientos que estén alineados con la Ley 21.663 Marco de Ciberseguridad.

Por esta razón, se recomienda seguir estándares que guíen la implementación de procesos de autenticación y autorización, como el uso correcto de JWT, procesos de intercambio y verificación de tokens, como asegurar *claims* válidos, especificación para contraseñas y protocolos de comunicación segura. Además, seguir estándares para implementar una buena política de protección de datos, incluyendo hashing, cifrado de datos, políticas de privilegio mínimo en los microservicios, desacoplamiento de datos (anonimizar al usuario real de los datos georreferenciados); y establecer un ciclo de vida para los datos, como el borrado de datos personales cuando un usuario es removido o el archivado o destrucción de archivos.

4.3.2 Rendimiento y Escalabilidad

Para la viabilidad operativa en entornos de producción, es importante asegurar un nivel óptimo de rendimiento y alta disponibilidad. Esto requiere una arquitectura diseñada para la escalabilidad horizontal, por lo tanto, es importante priorizar un diseño *stateless* y desacoplado en los microservicios. Con esto, la implementación de orquestación automatizada de contenedores (a través de Kubernetes, por ejemplo) será más fácil, y permitiría al sistema a reaccionar dinámicamente ante variaciones en cargas de trabajo.

Los contextos centrales del negocio, que incluyen la gestión de eventos y la autenticación de usuarios deberían mantener alta disponibilidad, ya que estos constituyen la ruta crítica operativa. En otras palabras, si estos servicios no están disponibles, se bloquearía la funcionalidad principal del sistema. Para proteger estos servicios, se recomienda implementar patrones de resiliencia como *Circuit Breakers* y *Timeouts* en las API principales. La idea es que fallos transitorios o latencias elevadas no degraden la experiencia de usuario final.

Por otro lado, los contextos intensivos en recursos como el almacenamiento de medios, la visualización de mapas y el procesamiento de datos, pueden ser un potencial cuello de botella. Para mitigar esto, la arquitectura debe delegar estas operaciones a procesos asíncronos mediante un bróker de mensajería como RabbitMQ. Esta estrategia de desacoplamiento busca que la latencia inherente de estas tareas no bloquee el flujo principal de interacción con el usuario.

Finalmente, para asegurar que el rendimiento se mantenga estable a lo largo del ciclo de vida del software, se puede considerar la integración de estrategias de Aseguramiento de Calidad (QA). Esto incluye la implementación de pipelines de CI/CD automatizadas y entornos de pruebas (unitarias y de integración), permitiendo detectar regresiones de rendimiento y errores de forma temprana antes del despliegue en producción.

4.3.3 Observabilidad

En una arquitectura distribuida, es importante considerar la observabilidad del sistema. Este concepto nos guía a prácticas que nos permiten conseguir una vista completa del sistema y los servicios internos. El objetivo de la observabilidad es identificar fallos de forma temprana, lograr trazabilidad de los datos, las operaciones, y facilitar la corrección de errores.

Para este requerimiento, es crucial el uso correcto de IDs únicos tanto para cada evento, archivo, usuario, dispositivo y operación. El uso de IDs de correlación es una ayuda, ya que nos permite vincular cada operación del sistema a través de diferentes servicios. También se recomienda el uso de soluciones de *logging* y centralización de registros (como la combinación Grafana-Loki), con tal de mantener historial y poder analizarlo con herramientas, incluso automatizadas. Cada servicio debería registrar operaciones CRUD, estados de procesamiento, fallos, excepciones, y tiempos de respuesta.

También se debe registrar operaciones críticas del servicio de autenticación, por ejemplo: inicios de sesión erróneos consecutivos, cambios de contraseña, bloqueos de cuentas y conexiones IP reiteradas. Es necesario considerar que los *logs* no deben incluir información sensible, y en el caso que sea inevitable, deberían ser enmascarados, es decir, evitar que dicho dato sea asociable a un usuario o evento específico. Por ejemplo: *hasheando* los IDs de usuario, o reduciendo la precisión de coordenadas.

4.4 Implementación

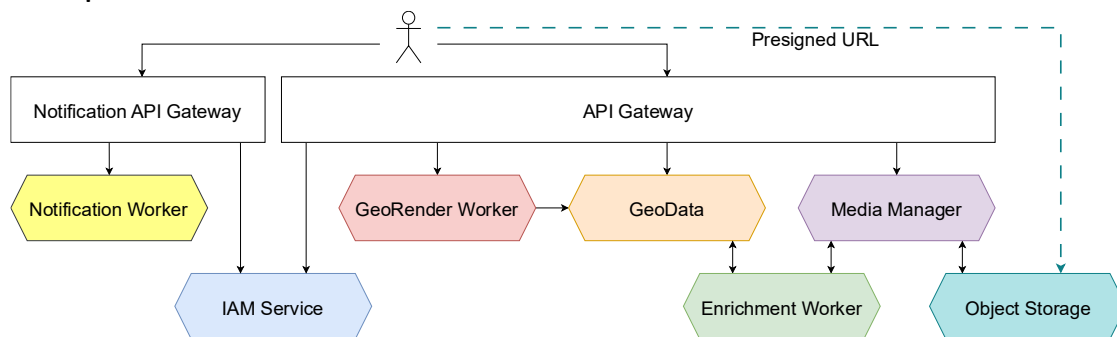


Figura 4. Propuesta de implementación y relación entre microservicios. Fuente: elaboración propia

Se considera una arquitectura basada en microservicios detrás de un API Gateway, véase la Figura 4.

Existen dos: 1) **API Gateway** principal, destinado al enrutamiento, autenticación y autorización de las consultas hacia el resto de los microservicios internos; y 2) **API Gateway de notificación**, destinado a mantener y gestionar las conexiones con el servicio de notificaciones. El primero requiere mantener disponibilidad, es central en el negocio, y está construido con un diseño sin estado que permite poder escalarlo sin problema

en un orquestador. El segundo está pensado específicamente para gestionar y mantener las conexiones con el servicio de notificaciones, por lo que necesita mantener un estado interno y otras estrategias de escalado.

Ambos gateways utilizan el servicio **IAM** (identidad y acceso), quien tiene la responsabilidad de mantener la autenticación y autorización de los usuarios. Genera tokens de acceso, verifica roles, y autoriza dispositivos. El resto de microservicios se comunican internamente con este servicio para verificar los tokens de acceso.

El servicio **GeoData** está destinado a la gestión de eventos. Este es el servicio central del negocio, la funcionalidad del usuario y la fuente de verdad: almacena todos los metadatos y la información geográfica de los eventos en formato JSON. Mientras que **GeoRender** se puede considerar a grandes rasgos como un servicio de agregación: se encarga de recopilar información y presentarla en un formato específico. Internamente se comunica con **GeoData** para obtener los datos sobre los eventos, y con ello, poder presentarlos al cliente en un formato adecuado para mapas, ya sea compilando directamente Vector Map Tiles pre-anotados o colecciones GeoJSON preprocesados para que el cliente los muestre. **GeoData** y **GeoRender** separan las necesidades de escritura y lectura específicamente.

Enrichment Worker está destinado al enriquecimiento de datos, y para efectos de esta propuesta, se abre a libertad de implementación. Se considera como un servicio genérico que puede abordar múltiples formas y herramientas para funcionar, por ejemplo: un modelo de IA, algoritmos de visión computacional, analítica o alguna herramienta de automatización.

Es ideal que el funcionamiento de los servicios **GeoRender** y **Enrichment Worker** sea fundamentalmente asíncrono. En otras palabras, no debería ser necesario una interacción directa con esos servicios, sino que deberían poder actuar como *workers* autónomos: ser capaces de realizar trabajos en segundo plano de manera automática. Para ello, se reitera la importancia de implementar y usar un bróker de mensajería de forma correcta.

Media Manager y **Object Storage** son servicios que se encargan del dominio de los archivos. **Media Manager** administra la lógica de negocio, gestiona, y delega la manipulación de binarios al **Object Storage**. Este último se encarga de manipular y servir los archivos binarios. Todo acceso al **Object Storage** debe ser autorizado previa y exclusivamente por el servicio **Media Manager** únicamente a través de URL pre-firmadas temporales y de un sólo propósito (carga/descarga). Se propone el uso de MinIO como **Object Storage**.

Notification Worker es un servicio opcional para la difusión de notificaciones del sistema. Está pensado principalmente para aportar a la experiencia de usuario. Casos de uso pueden ser, por ejemplo: notificar al cliente móvil que los datos enviados están siendo procesados o notificar al cliente web de un nuevo evento y que la interfaz reaccione acorde. Se basa fundamentalmente en escuchar asíncronamente a otros servicios por mensajes que requieran ser propagados.

4.5 Comunicación entre servicios

La siguiente sección busca especificar que métodos de comunicación utilizar entre los microservicios, además de las consideraciones al tomar dicha decisión.

4.5.1 Interacción síncrona

Se espera comunicación síncrona en los procesos que involucren directamente al cliente, por consultas de escritura/lectura de eventos, subida/descarga de archivos y autenticación. Este estilo de comunicación debería ser implementada usando REST con un modelo *request-response*.

Los metadatos de los eventos que maneja el servicio GeoData deberían ser cadenas de texto plano. Por esta razón, como consideraciones de rendimiento, la latencia de red tendrá más relevancia que el uso de transferencia de datos. Además, se busca mantener una consistencia *read-your-writes* con este servicio. La idea es que apenas el cliente capture un evento y envíe los metadatos, el servicio pueda guardarlo y servirlo inmediatamente. Así, consultas de lectura posteriores siempre retornarán la versión más actualizada de la base de datos.

Consultas hacia el servicio IAM son síncronas por necesidad: se requiere una autorización antes de poder interactuar con el resto del sistema y sus servicios. De la misma forma, Media Manager también requiere comunicación síncrona, ya que es necesario solicitar URL pre-firmadas antes de obtener los archivos.

Por otro lado, el servicio Enrichment Worker comunica sus resultados de manera síncrona con el resto de servicios, pero sus tareas de análisis son iniciadas de forma asíncrona. De forma similar, las consultas de lectura hacia GeoRender son síncronas, pero las operaciones de escritura, asíncronas. Véase a continuación.

4.5.2 Interacción asíncrona

Se espera comunicación asíncrona en los procesos que involucren programar tareas, realizar procesos intensivos en recursos (como IA), recopilación y agrupación de datos. Este estilo de comunicación debería ser implementado usando un bróker de mensajes.

Los servicios GeoData y Media Manager notifican al resto de servicios cada vez que hay nuevos eventos o archivos, con el fin de que el servicio Enrichment Worker pueda programar una tarea de análisis sobre estos.

El servicio debería ser capaz de consumir la cola de mensajes, y realizar tareas de análisis a demanda o en forma de lotes. La decisión sobre cada cuánto se deben realizar estas tareas debe tener en consideración el tiempo que demora en obtenerse los resultados, y que tan inmediata debe ser la disponibilidad de los resultados. Estos últimos son enviados de manera síncrona a GeoData y Media Manager.

De la misma forma, el servicio GeoRender también debe consumir la cola de mensajes, específicamente aquellos que GeoData envía. Con esta información, la idea es que, sin necesidad de consultas adicionales, GeoRender pueda registrar el evento con su propio formato, ya sea en forma de Vector Map Tiles o GeoJSON. Posteriormente, toda esa información ya compilada se sirve de forma síncrona a los clientes, como se mencionó en la sección 4.5.1.

4.6 Flujos lógicos principales

A continuación, se presenta algunas recomendaciones del flujo lógico que deberían seguir los procesos más fundamentales del sistema. Esta sección no busca ser una guía estricta de implementación, sino demostrar la interacción y el funcionamiento esperado de los microservicios.

4.6.1 Autenticación de usuarios

- 1) Un cliente realiza una consulta al API Gateway sin credenciales, por lo que le responde con error 401 (no autorizado).
- 2) Se solicita iniciar sesión o registrarse. El servicio IAM almacena los usuarios en una base de datos relacional interna. La contraseña es almacenada criptográficamente utilizando Argon2id (RFC 9106) (véase Seguridad).
- 3) La autenticación se realiza usando JWT con firma asimétrica RS256: una clave privada asociada a una clave pública identificado con un “kid” (key ID). La clave pública será guardada y expuesta en un endpoint. El API Gateway y el resto de los servicios pueden comunicarse internamente con el servicio IAM para consultar y verificar la autenticidad de los tokens generados.
- 4) Para autorizar dispositivos móviles específicos, se propone el uso de DPoP. Cada dispositivo tendrá un par de claves pública/privada, y el sistema verifica que el token enviado desde el cliente móvil efectivamente viene desde un dispositivo autorizado. Los IDs de los dispositivos móviles van asociados a un usuario y son almacenados en una tabla diferente.

La verificación del token en el API Gateway se considera la primera barrera de entrada al sistema (verificación de borde). Si se desea implementar una política *zero-trust*, se espera que los microservicios implementen verificación de token con el servicio IAM, para así evitar spoofing interno, y autorizar acciones. Para evitar tener que consultar a cada momento al servicio IAM para verificar la firma del JWT, se recomienda implementar un caché en el API Gateway que almacene temporalmente la clave pública. Para almacenar el JWT en una cookie, se recomienda aplicar los atributos “HttpOnly” y “SameSite”. [40]

Aunque DPoP añade complejidad en los clientes al tener que lidiar con pares de claves, se elige este estándar por sobre otros mecanismos de autorización de dispositivos más simples (como añadir un *claim* “device_id” en el JWT). La razón de esta decisión nace principalmente por la naturaleza distribuida de esta nueva arquitectura. DPoP previene criptográficamente el robo y la retransmisión de tokens: los Access Token quedan atados a la clave privada del dispositivo. Por lo tanto, ante un incidente de robo de tokens, el riesgo por daños se ve disminuido en gran medida, apoyando así el requerimiento de seguridad por datos sensibles. Para mitigar la posible sobrecarga adicional a causa de los cálculos criptográficos adicionales, se recomienda exigir DPoP únicamente en consultas que se requiera permisos de dispositivo.

4.6.2 Ingesta de eventos y archivos

- 1) El cliente en el dispositivo móvil captura un evento y realiza una consulta para guardar los datos. El API Gateway se comunica internamente con los servicios GeoData y Media Manager de ser necesario.
- 2) Los metadatos se envían por REST al servicio GeoData. Este genera un ID para el nuevo evento y luego guarda los metadatos en su base de datos interna. Se confirma el guardado al cliente respondiendo con dicha ID. Al mismo tiempo, GeoData emite un mensaje asíncrono al resto de servicios sobre el nuevo evento guardado.
- 3) De ser necesario, luego de que el cliente reciba la ID del nuevo evento, el cliente realiza una consulta para subir un archivo asociado a dicho evento. Esta consulta realizará una solicitud al servicio Media Manager para obtener una URL pre-firmada desde Object Storage. Con dicha URL, el cliente puede cargar archivos directamente al Object Storage, en una ubicación específica asociado a dicho evento.
- 4) Una vez el archivo se haya cargado, el Object Storage emite un mensaje a Media Manager para confirmar la carga del nuevo archivo, para que lo registre en su base de datos interna. Al mismo tiempo, este último emite un mensaje asíncrono hacia el resto de servicios sobre el nuevo archivo listo.

Un diagrama de secuencia de esta descripción se encuentra en el Anexo B. Si es necesario transferir imágenes a través de red interna o entre servicios, se recomienda codificar los archivos en Base64.

4.6.3 Procesamiento asíncrono

El servicio Enrichment Worker actúa de manera asíncrona e independiente. Realiza análisis en los momentos que se estime conveniente, por ejemplo: a través de tareas periódicas programadas (*cron jobs*), *on-demand* (analizar inmediatamente apenas lleguen datos), entre otras estrategias. Luego de los análisis, retorna mensajes sobre los resultados hacia el resto de servicios. Véase el Anexo C.

- 1) El servicio está a la escucha de mensajes “nuevo evento” y/o “nuevo archivo”; dependiendo de los requerimientos del negocio, se puede esperar por uno o ambos eventos antes de comenzar el procesamiento.
- 2) Luego, el servicio consulta internamente a GeoData o a Media Manager según los datos que se requieran para analizar.
 - a) Si se requiere metadatos, el servicio consulta directamente a GeoData.
 - b) Si se requiere uno o más archivos, el servicio debe consultar a Media Manager por una URL pre-firmada de descarga, y luego, guardar dichos archivos internamente de manera temporal mientras se procesa.
- 3) Con los datos recibidos, el servicio realiza el análisis.
- 4) Una vez listo, los resultados que implican modificar o añadir metadatos son emitidos a GeoData. Los resultados que son archivos deben ser guardados en el Object Storage, asociado al mismo evento, por lo cual debe pedir una URL pre-firmada de carga a Media Manager.
 - a) Cuando GeoData reciba dichos datos, este servicio debe emitir un mensaje al resto de servicios anunciando que un evento fue modificado.
 - b) Por otro lado, cuando Object Storage guarde el nuevo archivo, Media Manager no debe anunciarlo al resto de servicios, de lo contrario, causaría un bucle de mensajes infinito.⁸

Para prevenir el bucle de mensajes, Media Manager debe implementar una política de discriminación de origen. En otras palabras, el servicio no debe anunciar el archivo que Enrichment Worker acaba de guardar.

⁸ En este caso, ocurre cuando los archivos resultantes del análisis disparan nuevamente el evento de 'nuevo archivo' en el *Media Manager*, provocando que el *Enrichment Worker* procese sus propios resultados indefinidamente.

Para lograr esto, se puede organizar el Object Storage en subcarpetas lógicas, por ejemplo “subidas” (para los usuarios) y “resultados” (para los análisis), de tal forma que los archivos nuevos que sean cargados solo en la carpeta “subidas” sean anunciados.

Para aumentar la resiliencia ante errores, se recomienda el uso de una base de datos que almacene el estado, progreso y archivos involucrados. Con esto, se busca que el servicio pueda retomar cualquier tarea pendiente después de reiniciarse. En el caso de un fallo en las tareas de procesamiento, se recomienda como mínimo, guardar la información relevante en una colección/tabla separada para un posterior tratamiento.

4.6.4 Compilación de eventos en mapas

El servicio GeoRender *Worker* actúa de manera asíncrona e independiente. Está a la escucha de mensajes desde el servicio GeoData cada vez que se guarda, modifica o elimina un evento. De la misma manera que Enrichment Worker, GeoRender puede trabajar en los momentos que se estime conveniente.

Para esto último, se recomienda una estrategia de *debounce buffer*: se establece dos temporizadores, uno corto que se reinicia cada vez que se recibe un dato, y otro más largo que actúa como temporizador global. Si uno de estos termina sin interrupciones, entonces inicia la compilación con todos los datos recibidos hasta el momento. La idea de esta estrategia es mantener un equilibrio entre rendimiento y “frescura” de los datos.

Una vez compilados, se guarda en su base de datos y se sirven de manera síncrona hacia los clientes a través del API Gateway. Los clientes usarán estos datos para presentar un mapa con los datos registrados.

4.7 Riesgos y desafíos

En esta sección se identifican los riesgos y desafíos inherentes a la transición desde un sistema monolítico hacia una arquitectura distribuida basada en microservicios. Si bien muchos de estos desafíos son intrínsecos al paradigma distribuido, su identificación temprana es crítica para fundamentar las decisiones estratégicas durante la implementación.

Este análisis cubre desde la complejidad operativa y el rendimiento en las consultas, hasta el cumplimiento normativo en ciberseguridad. Además, con el objetivo de garantizar la viabilidad del sistema, para cada riesgo se plantean estrategias de mitigación.

4.7.1 Complejidad operativa

Implementar una arquitectura distribuida implica una mayor complejidad en la manera de gestionar y comunicar diferentes servicios. Esto incluye el aumento en el costo de desarrollo, mantenimiento y depuración.

Usar un bróker de mensajes y comunicación asíncrona en general añade complejidad en la trazabilidad y consistencia de las operaciones entre servicios. Si bien la asincronía permite mayor agilidad al no tener que esperar operaciones bloqueantes, la complejidad recae en la forma efectiva de manejar la asincronía: no perder la consistencia de los datos, lograr ubicar problemas de forma eficiente, y evitar dependencia con otros servicios. También, considerar aspectos de seguridad más robustos implica necesariamente un *trade-off* con el rendimiento de los procesos y la conveniencia, tanto para el desarrollador como para el usuario.

Para mitigar posibles problemas de operatividad:

- Utilizar soluciones de recopilación de registros y trazabilidad, por ejemplo, Grafana Loki. Aunque esto requiere implementar un nuevo servicio, estas soluciones de observabilidad son autocontenidos y estables. Pueden ser desplegados en un entorno ya existente.
- Si el negocio no requiere ciertos aspectos de seguridad de esta propuesta, por ejemplo “no hay necesidad de separar permisos de dispositivo específico”, mecanismos como DPoP pueden ser omitidos en la implementación.
- El escalado del servicio de notificaciones, al ser un servicio con estado, requiere estrategias adicionales para distribuir mensajes a instancias específicas, por ejemplo: *sticky sessions* (mantener una sesión unida a una instancia específica) o un bus de eventos interno.

4.7.2 Rendimiento en consultas espaciales

El núcleo de funcionalidad de este tipo de sistemas se basa en la georreferenciación de eventos urbanos. Por lo mismo, un desafío es: mantener eficiente y disponible los servicios responsables de estos datos. El impacto en la usabilidad desde la perspectiva del usuario se vería gravemente afectado si los servicios asociados a este contexto no están disponibles o son ineficientes.

Para mitigar posibles problemas de rendimiento:

- La operación más común de los servicios GeoData y GeoRender será principalmente lectura. Para evitar que la base de datos haga múltiples consultas consecutivas, se pueden implementar cachés.
- Utilizar motores de bases de datos que permitan indexación geoespacial nativa. Algunos ejemplos: PostgreSQL con PostGIS, o colecciones de documentos GeoJSON guardados en MongoDB con índices.
- Optimizar las consultas realizadas a GeoData y GeoRender, con tal de evitar resultados muy grandes. Es decir: no devolver todos los eventos de una ciudad, sino solo aquellos que están dentro del área visible de algún mapa, con un límite de zoom, o en un radio determinado de distancia.
- Evitar duplicados. Si un evento del mismo tipo está muy cerca de otro, y la fecha es cercana, muy probablemente sean el mismo fenómeno físico en la realidad. Una posible forma de evitar duplicados puede ser agrupando eventos: anexando un nuevo evento a uno ya existente, evitando crear una nueva entrada.

4.7.3 Acoplamiento

Aunque la arquitectura de microservicios busca reducir el problema del acoplamiento entre los contextos del negocio, es inevitable toparse en un caso donde estos pueden acoplarse. En esos casos, hay que gestionarlos debidamente.

Se puede provocar un acoplamiento lógico por contratos entre servicios. Por ejemplo, un cambio en la estructura de los datos de GeoData puede requerir actualizar otros servicios como GeoRender, Enrichment Worker o el API Gateway. Esto también puede ocurrir con los formatos de los mensajes asíncronos.

Otro punto crítico considerable, es la dependencia con la cola de mensajes. Si el bróker falla o se satura por *back-pressure*, entonces el rendimiento del sistema se puede ver afectado. Principalmente el mecanismo de posprocesamiento de eventos. Esto puede ocurrir si “entran” más eventos de los que se pueden procesar en un periodo de tiempo. Si bien las consultas a GeoData, GeoRender, Media Manager o autenticación no debería dejar de estar disponibles, las tareas que se ejecutan en segundo plano pueden verse degradado con tiempos de espera más largos.

Para mitigar posibles complicaciones de acoplamiento y dependencia:

- Se recomienda establecer la estructura de los contratos antes de implementar los servicios, y al hacerlo, seguir estándares de buenas prácticas para diseñar API REST. [41] A partir de allí, considerar modificaciones o extensiones según requiera el negocio.
- Considerar el uso de “prioridades” en el sistema de mensajería, dando importancia a tareas esenciales. Por ejemplo: dar más preferencia a las tareas de procesamiento que a las notificaciones.
- Para evitar llenar la memoria de las colas en caso de *back-pressure*, los mensajes solo deberían contener texto plano, información mínima necesaria, y no incluir binarios. Por ejemplo: desde GeoData hacia Enrichment Worker, incluir IDs de evento, archivos, u otra información relevante para el procesamiento, pero no el archivo propiamente tal. Enrichment Worker es quien debe tener la tarea de consultar al resto de servicios.

4.7.4 Costos de almacenamiento

Un gran desafío en el caso de que un sistema de este estilo se masifique, es la capacidad de almacenar los datos, especialmente los archivos binarios. Los datos en texto plano como JSON o almacenados en bases de datos pueden rondar desde unos pocos kilobytes hasta cientos de megabytes si se consideran millones de usuarios. Sin embargo, para esa misma cantidad de usuarios, el tamaño de almacenamiento aumenta considerablemente.

La funcionalidad de almacenar archivos depende fuertemente de las necesidades del negocio, pero de ser así, puede formar parte importante del contexto de la información geográfica del evento. Por lo tanto, el impacto en la usabilidad a raíz de un problema de almacenamiento puede ser potencialmente alta.

Para mitigar posibles problemas asociados al costo de mantenimiento:

- Se recomienda definir una política de gestión y ciclo de vida de archivos óptima, que incluya tareas de limpieza y compresión de archivos antiguos.
- En el caso de imágenes tomadas desde móvil, procurar establecer un límite de resolución. Es decir, no utilizar la resolución máxima que ofrece el dispositivo, sino limitarlo a, por ejemplo, 1080p o inferior.
- En caso de que, por necesidades técnicas, bajar la resolución no sea una solución razonable (por ejemplo, un algoritmo de procesamiento requiera imágenes con buena calidad), soluciones de compresión en disco podrían ser una alternativa.
- Para archivos generales: se recomienda establecer verificaciones de tipos de archivos y límites de tamaño. En otras palabras: si el negocio necesita trabajar únicamente con ciertos tipos de archivo, entonces se recomienda restringir las subidas únicamente a extensiones válidas, y limitar el tamaño de estos.

4.7.5 Costos de mantenimiento

Implementar microservicios conlleva el uso de diferentes técnicas y herramientas de desarrollo: contenedores y orquestadores, diferentes frameworks de API, motores de bases de datos, bróker de mensajes, soluciones de observabilidad, almacenamiento de archivos, por nombrar algunos.

El impacto humano implica conseguir un equipo de desarrollo y operaciones que conozca y tenga habilidades para trabajar con estas herramientas. Por otro lado, asumiendo que no se elige un despliegue local, sino utilizar una plataforma en la nube, un riesgo podría ser comenzar a depender de los proveedores de nube (como AWS, Google Cloud o Microsoft Azure). Los costos de mantener este sistema podrían aumentar considerablemente si el sistema sigue escalando.

Para mitigar posibles problemas de mantenimiento:

- Definir equipos de desarrollo con roles específicos: un equipo que vele por la plataforma (*Platform Team*) encargado de mantener los servicios, la orquestación, el bróker y la observabilidad del sistema; y otro equipo que vele por el desarrollo y las operaciones (*DevOps Team*).
- Reducir y mantener un *stack* tecnológico simple. Por ejemplo, si se planea utilizar modelos de IA basados en librerías de Python, seguir utilizando Python para backend; y si se busca crear una aplicación multiplataforma, se puede seguir con frameworks de Javascript para frontend.
- Asegurar una buena documentación desde el inicio. Se puede utilizar frameworks que permiten documentación automática, por ejemplo, FastAPI.
- Adoptar servicios en la nube para componentes *commodity* genéricos. Por ejemplo: un bróker de mensajes como PaaS, y de esta manera reducir carga de mantenimiento directo. Sin embargo, hay que tener en consideración las posibles latencias en red que puede introducir.

4.7.6 Resiliencia y tolerancia a fallas

La arquitectura propuesta fue diseñada teniendo en consideración que los servicios puedan ser desarrollados de manera *stateless* en lo posible, de esta forma, permitir escalarlos y así mejorar la disponibilidad y la resiliencia frente a fallos.

Como estamos frente a un sistema distribuido, los fallos deben ser considerados “la norma”, es decir, hay que asumir que en algún momento van a ocurrir, por lo que hay que prestar suma atención a la forma en como los servicios lidian con esas situaciones y se recuperan. Por ejemplo: fallos en tareas asíncronas, que causarían que los resultados queden pendientes de forma indefinida, o fallos en la red. Otra forma de impacto puede ser el reintento de consultas, provocando potencial retrabajo o datos duplicados.

Para mitigar este desafío y mejorar la resiliencia de los servicios:

- Tomar un rol “defensivo” en el desarrollo de código. En otras palabras, aplicar verificaciones y capturas de excepciones en todos los procesos. Por ejemplo: los servicios no deberían caer por un error de validación de datos, datos inexistentes o del estilo, sino aplicar fallbacks como valores por defecto en caso de no existir datos o campos vacíos.
- Guardar una lista de trabajos con su estado en los servicios que requieren hacer tareas asíncronas. Se espera que al reiniciar un *worker*, este pueda retomar automáticamente un trabajo pendiente, o como mínimo, volver a procesarlo desde el inicio.
- Utilizar IDs de idempotencia en consultas que podrían causar retrabajo o duplicados. Por ejemplo, notificaciones o mensajes asíncronos desde GeoData.

4.7.7 Cumplimiento legislativo

Como se mencionó anteriormente en la sección 4.3.1, los datos almacenados en este sistema van a ser potencialmente sensibles: coordenadas geográficas, archivos, e identificación de usuarios únicos. Por lo mismo, el desafío de cumplir con los requerimientos de seguridad debe ser el punto más importante que tener en mente si se quiere lanzar un producto de esta naturaleza al mercado real.

No solo porque el sistema va a estar lidiando con información confidencial, sino que además también es necesario alinearse con las obligaciones locales en donde el sistema será instalado. En otras palabras, hay que alinear los requerimientos de seguridad con el contexto legislativo en donde el sistema será utilizado. Para el caso de Chile, sistemas de este tipo se pueden considerar parte de un Servicio Esencial si es utilizado por instituciones públicas. [36] Con esto, se debe hacer efectivo el cumplimiento de la Ley 21.663 sobre Marco de Ciberseguridad, y especialmente la futura Ley 21.719 sobre Protección de Datos Personales (actualmente la Ley 19.628 sobre la Protección de la Vida Privada).

Medidas principales que se recomiendan para mitigar problemas relacionados a cumplimientos de seguridad:

- Considerar los requerimientos sobre seguridad y rendimiento expresadas en las secciones 4.3.1 y 4.3.2, con una implementación que apoye a la triada CIA. Por ejemplo: la lógica de autenticación de usuarios propuesta en la sección 4.6.1 haciendo uso de JWT+DPoP (Confidencialidad, acceso controlado y autorizado) y hashing y thumbprints (Integridad, autenticidad de los datos), así como implementar servicios sin estado (*stateless*) que permitan ser escalados (Disponibilidad, reducción de puntos únicos de fallo).
- El uso de servicios de observabilidad expresados en la sección 4.3.3 permiten tener trazabilidad de las acciones dentro de los servicios, y así, detectar posibles patrones anormales. Esto permite recopilar información para reportes de incidencias como lo exige la ley.
- Definir políticas sobre el ciclo de vida tratamiento de los datos alineados con los requerimientos de seguridad y los costos de almacenamiento. Incluyendo derechos ARCO, transparencia, y consentimiento sobre los datos personales, alineado con la Ley 19.628 sobre la Protección de la Vida Privada. [42]
- Implementar un SGSI (Sistema de Gestión de Seguridad de la Información) en la organización siguiendo especificaciones como ISO/IEC 27001. Organizando protocolos organizacionales enfocados en seguridad, incluyendo: enfoque de desarrollo basado en riesgos, ciclo de mejora PDCA (*Plan-Do-Check-Act*) y documentación de la Declaración de Aplicabilidad (SoA).

5 Validación

Para la validación de la propuesta, se realizó una implementación parcial de microservicios en el proyecto original de Geoviality. El objetivo de esta prueba es obtener métricas enfocadas en el rendimiento y utilización de recursos, y compararlos con las métricas obtenidas después de hacer los cambios.

Para este caso, se experimentó solo con el proceso de ingesta de eventos (véase la sección 4.6.2), y el procesamiento asíncrono (véase la sección 4.6.3). Las pruebas fueron hechas en un procesador AMD Ryzen 7 9700X de 16 hilos. El modelo de IA para el servicio de procesamiento es la misma del proyecto original (YO-LOv8). Cabe recalcar que una limitante del experimento es que el modelo no se ejecutó en GPU. Esto es porque

el entorno de prueba no poseía una tarjeta admisible para CUDA, por lo tanto, el *throughput* del modelo puede verse degradado. Para producción, se recomienda ejecutar el modelo en GPU en lo posible.

Para las métricas concretas, se toma en consideración las siguientes:

Tabla 5. Enumeración y definición de métricas

Nombre	Definición
Uso de almacenamiento	Diferencia en el espacio del disco duro utilizado por archivos.
Uso de red	Cantidad de datos que fluyen a través de la red en un determinado intervalo de tiempo.
Tiempo de procesamiento	Tiempo que tarda el servicio de IA en descargar y procesar un evento.
Tiempo de cola	Tiempo desde que un evento es ingresado en la cola hasta que es procesado por el servicio de IA. Representa el tiempo de cómputo total.
Tiempo de recepción	Tiempo que tarda un evento desde la consulta hasta ser registrado en la base de datos. El usuario percibe este tiempo de forma directa.
Máximo de mensajes en cola	Cantidad máxima de mensajes que han estado en cola. Para medir <i>back-pressure</i> en el bróker y el cuello de botella del servicio IA.

5.1 Grupo control

Como grupo control, se utilizó el proyecto original, y se realizó una prueba de rendimiento utilizando Postman con 10 usuarios virtuales realizando consultas simultaneas por 1 minuto de forma progresiva. La consulta es de ingesta de evento (véase la sección 3.2.1) y archivo, con una imagen de ≈600 [KB]. Se ejecutaron 334 consultas en total. Los resultados fueron los siguientes:

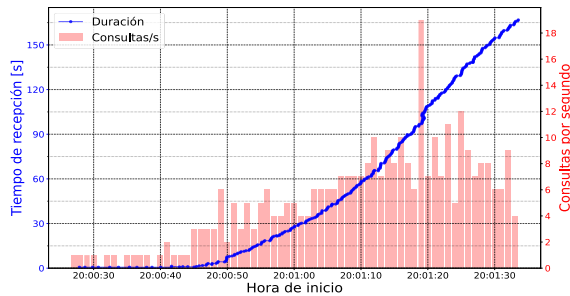


Figura 5. Tendencia del tiempo de recepción en relación con la cantidad de consultas/segundo. Fuente: elaboración propia.

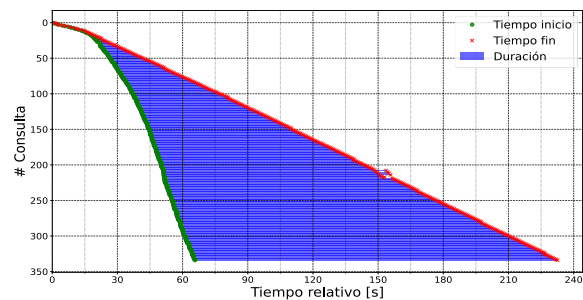


Figura 6. Gráfico estilo Gantt representando el tiempo de cola de cada consulta. Fuente: elaboración propia.

	API	IA	RabbitMQ
Uso de almacenamiento	202 [MB]	367 [MB]	≈370 [MB]
Uso de red	≈1.12 [GB]	≈860 [MB]	≈450 [MB]
Tiempo de procesamiento	n/a	≈300 [ms]	n/a
Máximo de mensajes en cola	n/a	n/a	≈250

Nótese en la Figura 5 cómo es que el tiempo de recepción (o tiempo percibido) aumenta a medida que se reciben más consultas. Esto ocurre porque el evento es ingresado a la base de datos **después** de que el servicio IA termine su análisis. A raíz de lo anterior, nótese en la Figura 6 que la última consulta (en la parte inferior) será siempre la que más tiempo va a estar esperando en la cola. Esto es un efecto del *back-pressure* de la cola de mensajes: llegan más imágenes de las que puede procesar simultáneamente el servicio IA, que actualmente es una a la vez. Por último, nótese el uso excesivo de datos en el almacenamiento y red. Esto es a raíz de la mala gestión y duplicación de datos tanto en la API como en el servicio IA, además de codificar los archivos completos dentro del mensaje que va hacia el bróker. Véase el Anexo F para más detalle.

5.2 Experimentación

Para la experimentación, se descompuso el Servidor IA en 2 microservicios parciales: GeoData (encargado de la lógica de negocio y la manipulación de la base de datos) y Enrichment Worker (encargado del

posprocesamiento asíncrono con IA). Se implementó el flujo de ingesta de eventos y procesamiento asíncrono definidos en las secciones 4.6.2 y 4.6.3. No se implementará el servicio de archivos; la API tomará este rol.

Se realizó una nueva prueba de rendimiento con los mismos parámetros anteriores. Con un total de 326 consultas, los resultados fueron los siguientes:

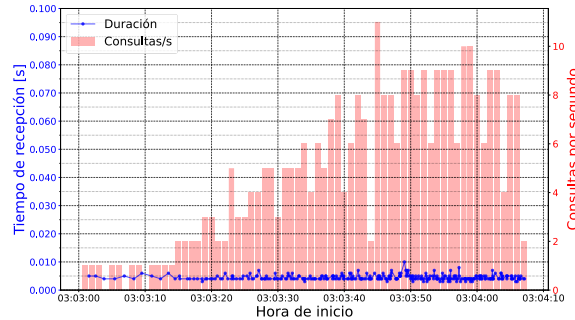


Figura 7. Tendencia del nuevo tiempo de recepción en relación con la cantidad de consultas/segundo. Fuente: elaboración propia.

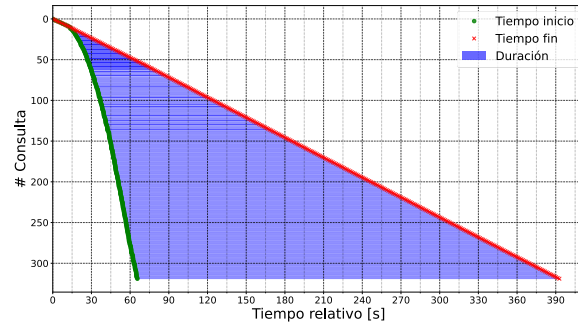


Figura 8. Gráfico estilo Gantt representando el nuevo tiempo de cola de cada consulta. Fuente: elaboración propia.

	API	Enrich. Worker	GeoData	RabbitMQ
Uso de almacenamiento	196 [MB]	0 [MB]	0 [MB]	≈2 [MB]
Uso de red	≈200 [MB]	≈200 [MB]	≈8.5 [MB]	≈600 [KB]
Tiempo de procesamiento	n/a	≈500 [ms]	n/a	n/a
Máximo de mensajes en cola	n/a	n/a	n/a	≈280

A comparación del grupo control, se observó una **disminución de ≈79% en el uso combinado total de almacenamiento** y una **disminución del ≈83% en el uso combinado total de red**. Esto se debe principalmente a la eliminación de la duplicidad y la redundancia de comunicación entre los servicios.

Sin embargo, en la Figura 8 se observa un incremento en el tiempo de tiempo de cola. Este fenómeno es esperado y corresponde a la sobrecarga introducida por la distribución del sistema: considerando que la imagen ya no viene incluida en el mensaje desde el bróker, el servicio Enrichment Worker debe ahora gestionar la descarga de archivos desde la API. No obstante, este costo adicional es despreciable a comparación de la ganancia obtenida en disponibilidad. En la Figura 7, se aprecia una **reducción del ≈99% en el tiempo de recepción** en la última consulta, pasando de casi 3 minutos a 4 milisegundos en promedio. Esto se debe al cambio a un modelo asíncrono; el cliente ya no tiene que esperar a que Enrichment Worker termine completamente para recibir una respuesta. Véase el Anexo G para más detalle.

Para reducir el tiempo de cola, recordemos que ahora los servicios API, Enrichment Worker y GeoData son microservicios independientes, sin acoplamiento, y sin estado. Esta nueva arquitectura nos permite replicar estos servicios sin problema. Para ello, se migraron los servicios a un clúster local de Kubernetes, con un deployment replicado para los servicios GeoData y Enrichment Worker. Se experimentó con 1 hasta 5 réplicas. Cabe destacar que, para esta migración, no fue necesario cambiar el código interno de los servicios.

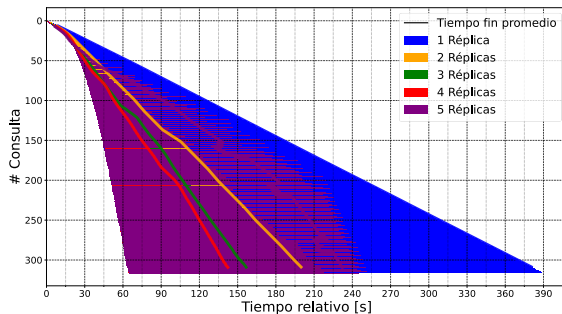


Figura 9. Gráfico estilo Gantt comparando el tiempo de cola respecto a la cantidad de réplicas. Fuente: elaboración propia.

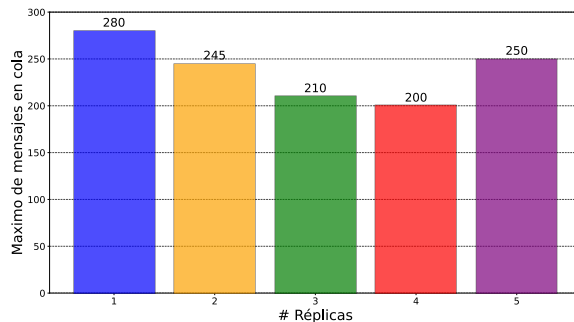


Figura 10. Cantidad máxima de mensajes en cola para cada cantidad de réplicas. Fuente: elaboración propia

Los resultados fueron los siguientes: métricas como el uso de almacenamiento, red, y tiempo de procesamiento no se vieron con cambios significativos, lo cual era esperable, ya que el uso de datos sigue siendo aproximadamente el mismo. Sin embargo, como se ve en la Figura 9 y Figura 10, los resultados muestran un gran cambio en el tiempo de cola y en la cantidad máxima de mensajes en cola. Nótese que, con 2 réplicas, el tiempo de cola se reduce prácticamente a más de la mitad.

No obstante, al llegar a las 5 réplicas, se observó una disminución de rendimiento general. Esto ocurrió porque al momento de la experimentación, el uso de la CPU de la máquina física llegaba constantemente al 100%, debido a la saturación de los núcleos físicos. En otras palabras, el hardware y la falta de recursos computacionales para soportar la cantidad de réplicas causaron un cuello de botella en el rendimiento. Se concluyó que, para este sistema en particular, el punto óptimo es de 4 réplicas, **disminuyendo un ≈64% el tiempo de cola** de la última consulta. Una transición a inferencia basada en GPU puede ser el siguiente paso para mejorar aún más el rendimiento del servicio de procesamiento, sin necesidad de escalar tantas réplicas.

6 Conclusión

El desarrollo de esta propuesta permitió cumplir con el objetivo principal de diseñar una arquitectura de software más genérica, robusta y extensible para sistemas de georreferenciación de eventos urbanos; tomando como base el proyecto original de Feria de Software y las falencias detectadas de este. A través de la aplicación de metodologías como *Domain-Driven Design*, se logró desacoplar la lógica de negocio original en diferentes contextos, facilitando la transición desde un sistema monolítico a una arquitectura de microservicios.

Los resultados obtenidos en la validación demostraron que el desacople parcial en microservicios mejoró la eficiencia técnica a comparación de la arquitectura original. Esta optimización interna permitió una reducción combinada del uso de almacenamiento en un 79% y del uso de red en un 83%. También expuso un *trade-off* clásico en sistemas distribuidos: se introdujo una leve sobrecarga en la eficiencia computacional total a cambio de ganar escalabilidad y alta disponibilidad. Si bien el ciclo completo de procesamiento (desde el envío hasta el resultado de la IA) aumentó levemente, la latencia percibida por el usuario (tiempo de recepción) se redujo en un 99%, pasando de minutos a milisegundos en situaciones de alta carga. Confirmado que la arquitectura propuesta prioriza correctamente la experiencia de usuario y la resiliencia del proceso de ingesta.

Con lo anterior, la nueva capacidad de escalado horizontal resultó ser una solución viable para la sobrecarga introducida por la arquitectura. Se usó un entorno de orquestación con Kubernetes para replicar los servicios GeoData y Enrichment Worker. Los resultados mostraron una reducción significativa del tiempo de cola de las tareas y del *back-pressure* de la cola de mensajes. Esto permitió análisis más ágiles y, al mismo tiempo, la disponibilidad del sistema sin bloquear la interacción con el usuario final.

En términos de seguridad y cumplimiento normativo, la propuesta eleva los estándares de calidad del proyecto, tomando aspectos que no fueron abordados en la implementación *ad hoc* original; transformando un prototipo académico a un sistema apto para entornos de producción real. La incorporación y definición de buenas prácticas para *JSON Web Tokens* (JWT) junto a *Demonstrating Proof-of-Possession* (DPoP), además de la alineación con requerimientos legislativos locales como la Ley Marco de Ciberseguridad y la Ley de Protección de Datos Personales, permiten implementar mecanismos de autenticación robustos y así, mitigar riesgos críticos de suplantación y manejo indebido de información sensible.

No obstante, también se observó que la adopción de una arquitectura distribuida conlleva un aumento inherente en la complejidad operativa. La necesidad de gestionar la comunicación asíncrona, la orquestación de contenedores y la implementación de servicios sin estado exigen un nivel adicional de madurez técnica a los desarrolladores, además del uso y configuración de herramientas de observabilidad superiores a las de un monolito. Por lo tanto, la implementación de esta arquitectura se justifica principalmente en escenarios donde se proyecta un alto volumen de transacciones, se necesita un escalado independiente de servicios, y se requiere el cumplimiento de estándares de seguridad.

En conclusión, la arquitectura diseñada no solo resuelve la deuda técnica heredada del proyecto original, sino que dota al sistema de la flexibilidad necesaria para evolucionar y adaptarse a múltiples contextos y requerimientos, estableciendo una base sólida y perdurable. Paralelamente, introduce un mayor nivel de

complejidad y requiere mayores consideraciones sobre el uso de recursos en hardware. Si bien los microservicios no son una solución universal, su implementación estratégica puede resultar en una gran ayuda para garantizar la mantenibilidad, escalabilidad y evolución a largo plazo de sistemas complejos.

Referencias

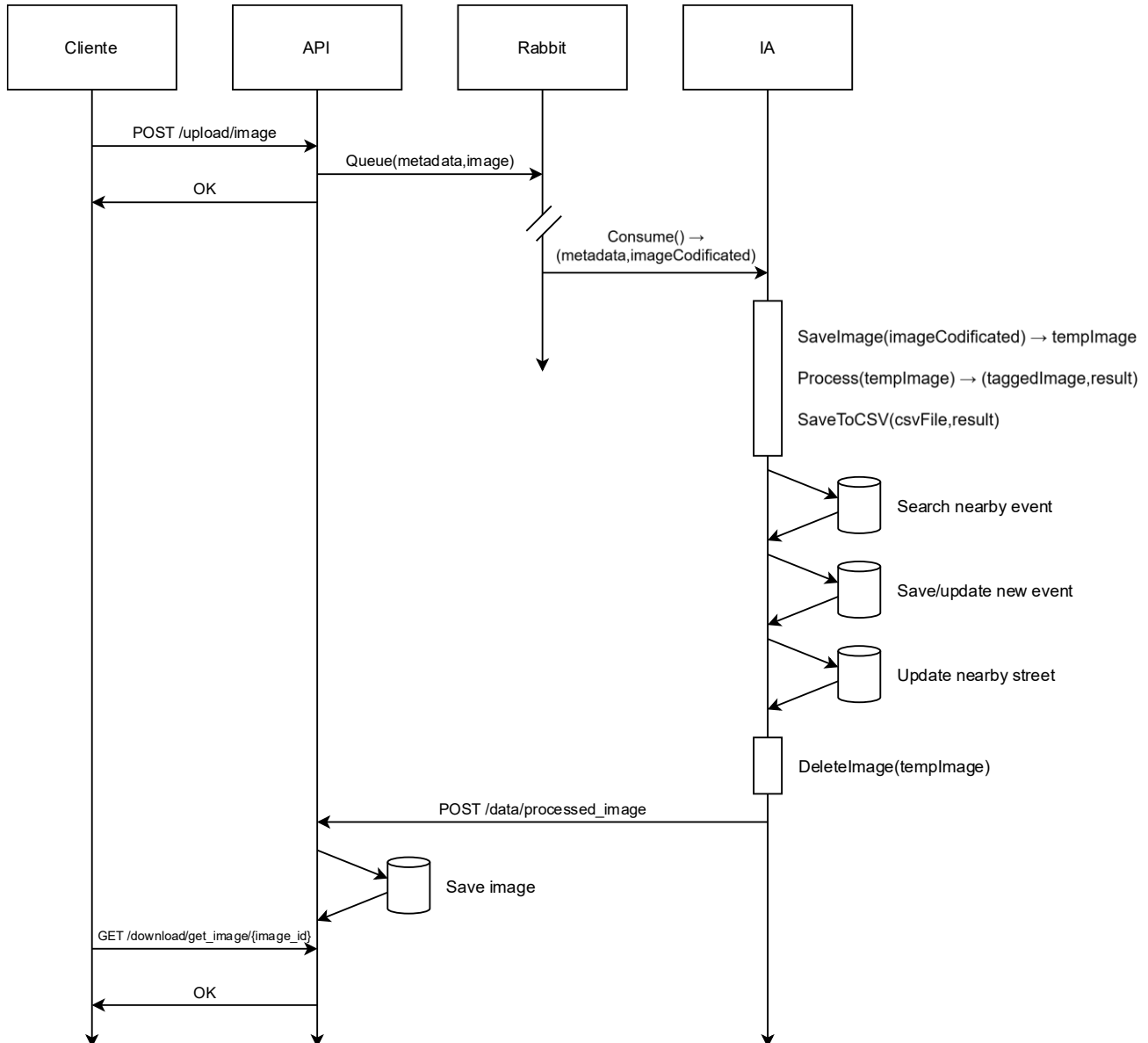
- [1] Departamento de Informática USM, “Estudiantes entrenan inteligencia artificial para detectar y clasificar irregularidades en la calzada”, 23 de octubre de 2024. Accedido: 4 de diciembre de 2025. [En línea]. Disponible en: <https://informatica.usm.cl/noticias/geoviality-xxxii-feria-de-software-usm/>
- [2] Diario El Mercurio, “Jueves de Innovación N° 313”, 12 de diciembre de 2024. Accedido: 11 de diciembre de 2025. [En línea]. Disponible en: https://www.linkedin.com/posts/innovacion-el-mercurio_juevesdeinnovaciaejn-ciencia-startup-activity-7272913167972810753-oZHW/
- [3] L. Bass, P. Clements, y R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.
- [4] M. Skelton y M. Pais, *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press, 2019.
- [5] “ISO/IEC/IEEE 42010:2022: Software, systems and enterprise — Architecture description”, nov. 2022. Accedido: 4 de diciembre de 2025. [En línea]. Disponible en: <https://www.iso.org/standard/74393.html>
- [6] R. C. Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Prentice Hall, 2017.
- [7] M. Richards y N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O’Reilly Media, 2020.
- [8] E. Gamma, R. Helm, R. Johnson, y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [9] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O’Reilly Media, 2021.
- [10] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O’Reilly Media, 2019.
- [11] M. Fowler y J. Lewis, “Microservices: A Definition of This New Architectural Term”, *martinfowler.com*, 2014, [En línea]. Disponible en: <https://martinfowler.com/articles/microservices.html>
- [12] S. Narwal y S. Sebastian, “Transform Monolith into Microservices using Docker”, *International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, jul. 2017, doi: 10.1109/ICCUBEA.2017.8463820.
- [13] D. Jaramillo, D. V. Nguyen, y R. Smart, “Leveraging microservices architecture by using Docker technology”, *Conference Proceedings - IEEE SOUTHEASTCON*, vol. 2016-July, jul. 2016, doi: 10.1109/SECON.2016.7506647.
- [14] B. Burns, J. Beda, K. Hightower, y L. Evenson, *Kubernetes: Up and Running*, 3rd ed. O’Reilly Media, 2022.
- [15] C. Henríquez, J. Derley, R. Valencia, y G. Sánchez, “Architectural Evolution from Monolithic to Microservices in Scalable Systems: A Case Study of Netflix”, *Prospectiva*, vol. 23, n° 1, mar. 2025, doi: 10.15665/RP.V23I1.3683.
- [16] O. C. Oyeniran, A. O. Adewusi, A. G. Adeleke, L. A. Akwawa, y C. F. Azubuko, “Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability”, *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*, vol. 1, n° 2, pp. 92–106, jul. 2024, doi: 10.61359/11.2206-2409.
- [17] M. L. Abbott y M. T. Fisher, *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley, 2015. Accedido: 5 de diciembre de 2025. [En línea]. Disponible en: <https://www.oreilly.com/library/view/art-of-scalability/9780134031408/>
- [18] G. Hohpe y B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.
- [19] A. S. Tanenbaum y M. van Steen, *Distributed Systems: Principles and Paradigms*, 3rd ed. CreateSpace Independent Publishing Platform, 2017.

- [20] H. Nejati, S. Aldin, H. Deldari, M. H. Moattar, y M. R. Ghods, “Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications”, feb. 2019, Accedido: 5 de enero de 2026. [En línea]. Disponible en: <https://arxiv.org/pdf/1902.03305>
- [21] J. Greenfield y K. Short, “Software factories”, en *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA: ACM, oct. 2003, pp. 16–27. doi: 10.1145/949344.949348.
- [22] R. F. Tomlinson, *Thinking about GIS: Geographic Information System Planning for Managers*, Third Edition., n° 1. ESRI Press, 2008. Accedido: 5 de diciembre de 2025. [En línea]. Disponible en: https://books.google.com/books/about/Thinking_about_GIS.html?hl=es&id=X8XgSAJrJVUC
- [23] P. A. Longley, M. F. Goodchild, D. J. Maguire, y D. W. Rhind, *Geographic Information Science and Systems*, 4th ed. Wiley, 2015.
- [24] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, y T. Schaub, “IETF RFC 7946: The GeoJSON Format”, ago. 2016, doi: 10.17487/RFC7946.
- [25] Mapbox, “Mapbox Vector Tile Specification”, 2025. [En línea]. Disponible en: <https://github.com/mapbox/vector-tile-spec>
- [26] PostGIS Project Steering Committee, “PostGIS: Spatial and Geographic Objects for PostgreSQL”. [En línea]. Disponible en: <https://postgis.net/>
- [27] M. F. Goodchild, “Citizens as Sensors: The World of Volunteered Geography”, *GeoJournal*, vol. 69, n° 4, pp. 211–221, 2007, doi: 10.1007/s10708-007-9111-y.
- [28] Docker Documentation, “What is Docker?” Accedido: 5 de diciembre de 2025. [En línea]. Disponible en: <https://docs.docker.com/get-started/docker-overview/>
- [29] Kubernetes Documentation, “Overview”. Accedido: 5 de diciembre de 2025. [En línea]. Disponible en: <https://kubernetes.io/docs/concepts/overview/>
- [30] R. M. Blank y P. D. Gallagher, “NIST SP 800-30: Guide for Conducting Risk Assessments”, sep. 2012, doi: 10.6028/NIST.SP.800-30R1.
- [31] A. Shostack, *Threat Modeling: Designing for Security*. Wiley, 2014.
- [32] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 20th Anniversary. John Wiley & Sons, 2015.
- [33] M. Jones, J. Bradley, y N. Sakimura, “IETF RFC 7519: JSON Web Token (JWT)”, 2015. [En línea]. Disponible en: <https://datatracker.ietf.org/doc/html/rfc7519>
- [34] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, y D. Waite, “IETF RFC 9449: OAuth 2.0 Demonstrating Proof-of-Possession (DPoP)”, 2023. [En línea]. Disponible en: <https://datatracker.ietf.org/doc/html/rfc9449>
- [35] B. Campbell, M. Jones, N. Sakimura, y F. Skokan, “JSON Web Token Claims”, Internet Assigned Numbers Authority (IANA). Accedido: 12 de diciembre de 2025. [En línea]. Disponible en: <https://www.iana.org/assignments/jwt/jwt.xhtml>
- [36] Biblioteca del Congreso Nacional (BCN), *Ley 21663 - Marco de Ciberseguridad*. Ministerio del Interior y Seguridad Pública de Chile, 2025. Accedido: 14 de diciembre de 2025. [En línea]. Disponible en: <https://www.bcn.cl/leychile/navegar?idNorma=1202434>
- [37] Biblioteca del Congreso Nacional (BCN), *Ley 21719 - Regula la Protección y el Tratamiento de los Datos Personales [...]*. Ministerio Secretaría General de la Presidencia de Chile, 2025. Accedido: 14 de diciembre de 2025. [En línea]. Disponible en: <https://www.bcn.cl/leychile/navegar?idNorma=1209272>
- [38] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.
- [39] D. L. Parnas, “On the criteria to be used in decomposing systems into modules”, *Commun. ACM*, vol. 15, n° 12, pp. 1053–1058, 1972, doi: 10.1145/361598.361623.
- [40] MDN Docs, “Using HTTP cookies”. Accedido: 29 de enero de 2026. [En línea]. Disponible en: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies#security>
- [41] V. Sahni, “Best Practices for Designing a Pragmatic RESTful API”. Accedido: 6 de enero de 2026. [En línea]. Disponible en: <https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>
- [42] Biblioteca del Congreso Nacional (BCN), *Ley 19628 - Sobre Protección de la Vida Privada*. Ministerio Secretaría General de la Presidencia de Chile, 2022. Accedido: 11 de enero de 2026. [En línea]. Disponible en: <https://www.bcn.cl/leychile/navegar?idNorma=141599>

Anexos

Anexo A

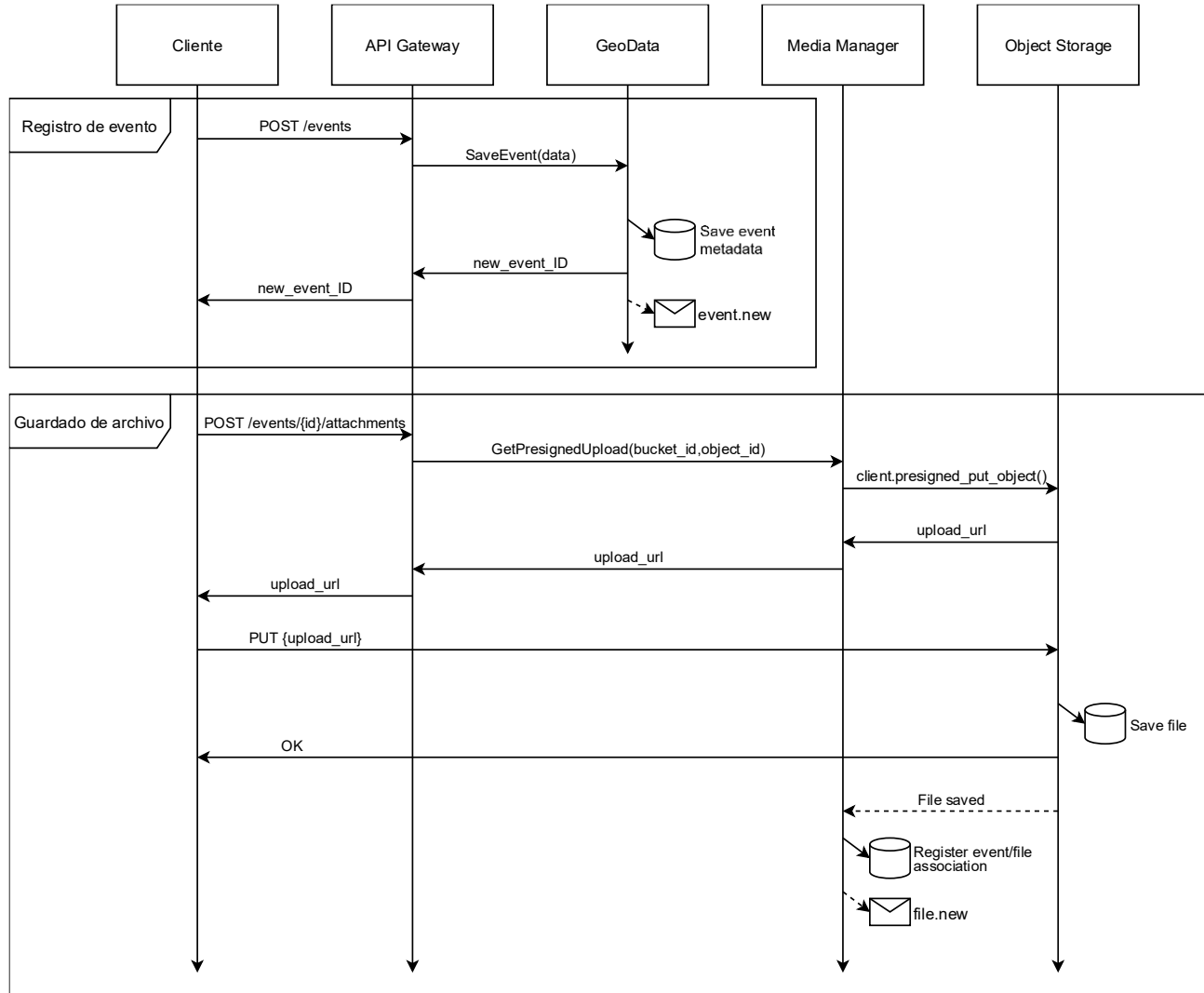
Análisis As-Is. Registro de datos y procesamiento. Diagrama de secuencia.



Anexo B

Propuesta To-Be. Ingesta de eventos y archivos. Diagrama de secuencia.

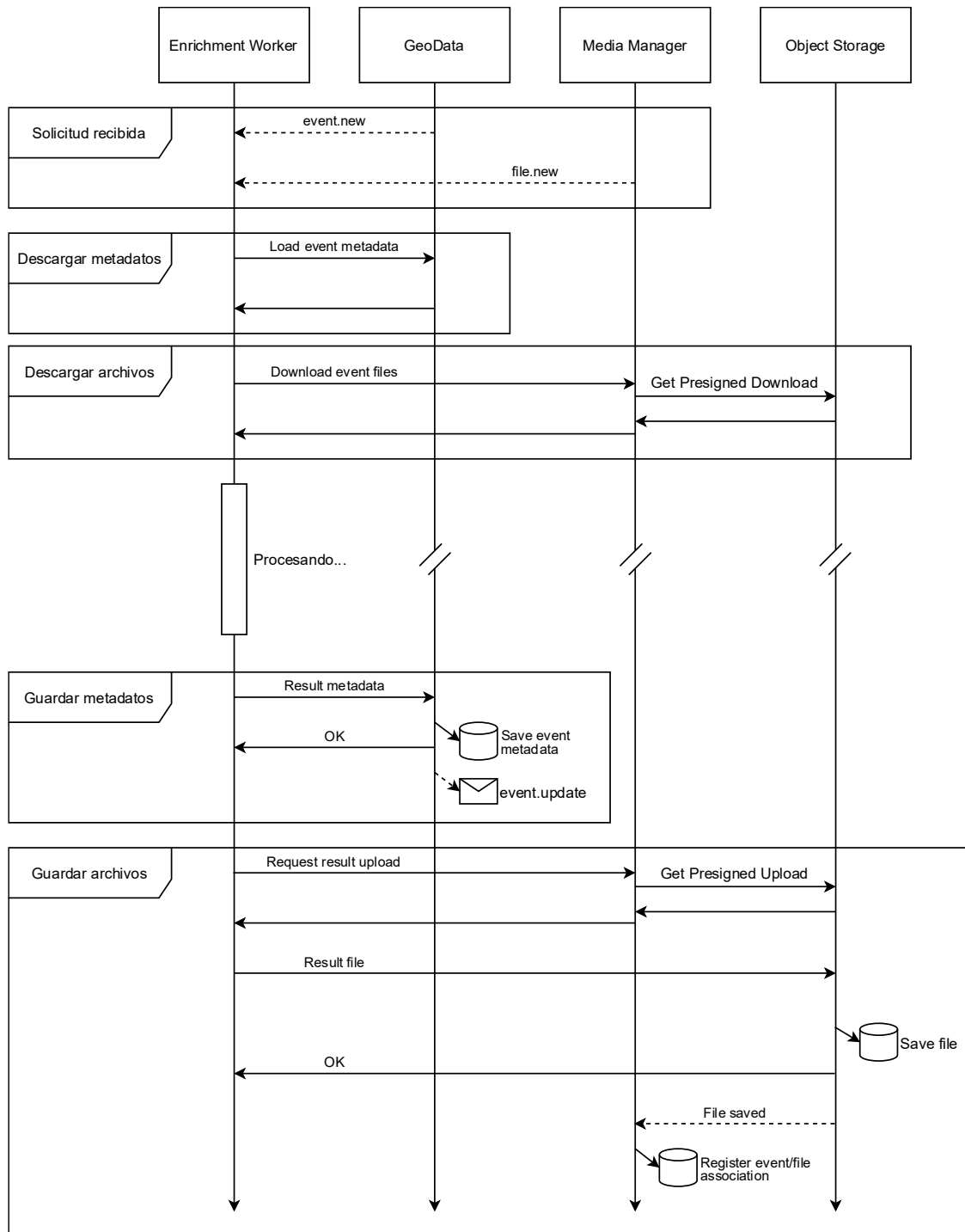
Nótese que a diferencia del proceso original en el Anexo A, el registro de eventos y el guardado de archivos son dos procesos diferenciados, iniciados por el cliente, y delegados a los microservicios correspondientes.



Anexo C

Propuesta To-Be. Procesamiento asíncrono. Diagrama de secuencia.

Nótese que a diferencia del proceso original en el Anexo A, el servicio Enrichment Worker tiene la responsabilidad de conseguir los datos necesarios por cuenta propia. Además, cada “paso” es diferenciable. Esto puede ser usado para mantener una base de datos de “trabajos pendientes”.



Anexo D

Demostración de prueba de posesión (DPoP). Diagrama de secuencia del funcionamiento básico. Obteniendo un Access Token “atado” a un dispositivo presentado en pseudocódigo.



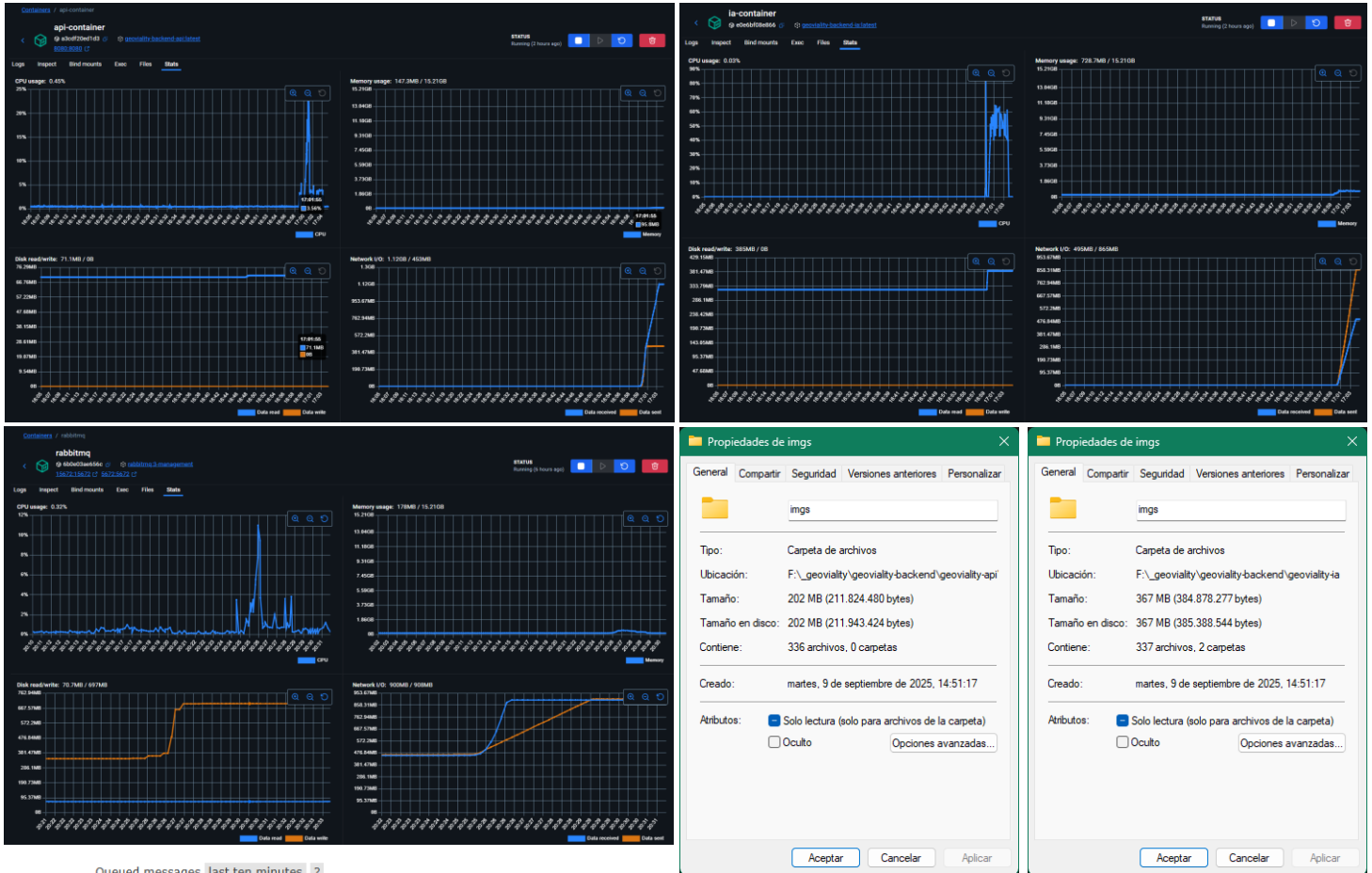
Anexo E

Demostración de prueba de posesión (DPOP). Diagrama de secuencia del funcionamiento básico.
Utilizando un Access Token ya adquirido junto a una prueba de posesión para acceder a un recurso protegido presentado en pseudocódigo.

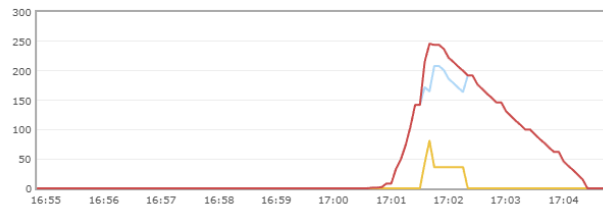


Anexo F

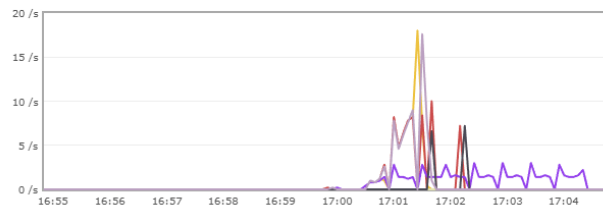
Métricas obtenidas en el grupo control, antes de realizar cambios experimentales de validación.



Queued messages last ten minutes ?



Message rates last ten minutes ?



Data rates last ten minutes

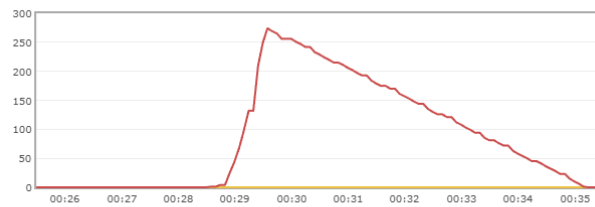


Anexo G

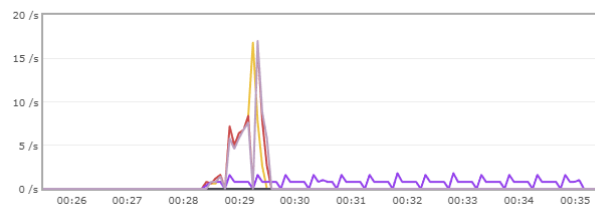
Métricas obtenidas en el grupo de experimentación, después de realizar los cambios en la arquitectura.



Queued messages last ten minutes ?



Message rates last ten minutes ?



Data rates last ten minutes

