



Diagnóstico y plan de mejora de la calidad de software en un sistema de gestión vial automatizado: Caso Geoviality

Felipe Ignacio Mellado Olea

felipe.melladoo@usm.cl

Profesor guía: Marcello Visconti Zamora

Resumen: Geoviality es un sistema de gestión vial automatizado cuyo backend, desarrollado inicialmente como prototipo académico bajo restricciones de tiempo, acumuló una alta deuda técnica que comprometía su seguridad y escalabilidad. El objetivo de este trabajo fue sanear dicha deuda mediante una estrategia de refactorización incremental basada en el modelo ISO/IEC 25010:2023, buscando transformar la arquitectura monolítica en una solución modular y robusta. La metodología aplicó herramientas de análisis estático y dinámico (SonarQube, OWASP ZAP, Bandit, Pylint y OWASP Dependency-Check) para el diagnóstico, validando la integridad funcional mediante un protocolo de pruebas de regresión manuales en flujos críticos y verificación de contratos API, estableciendo con ello las bases arquitectónicas necesarias para la futura automatización de pruebas. Los resultados evidencian la amortización total de la deuda técnica de código estimada por las herramientas, la eliminación de vulnerabilidades críticas y una reducción del 28% en la complejidad cognitiva. Se concluye que la intervención elevó la madurez tecnológica del sistema, evolucionando de un prototipo experimental a un Producto Mínimo Viable preparado para transferencia tecnológica.

Palabras Clave: Deuda Técnica, Refactorización de Software, ISO/IEC 25010:2023, Arquitectura de Backend, Aseguramiento de Calidad.

1 Introducción

El mantenimiento de la infraestructura vial es una condición necesaria para la seguridad y eficiencia del transporte urbano. Sin embargo, en la práctica, la detección y gestión de baches, grietas u otras irregularidades suele realizarse mediante inspecciones visuales y reportes ciudadanos, procesos que resultan lentos, poco trazables y costosos. Frente a este escenario, han surgido propuestas que utilizan visión por computador, inteligencia artificial y georreferenciación para automatizar la captura y análisis del estado de las calzadas.

En este contexto se sitúa **Geoviality**, un sistema de gestión vial inteligente desarrollado inicialmente bajo el marco de la Feria de Software de la Universidad Técnica Federico Santa María[1]. La solución integra una aplicación móvil, una plataforma web y un backend encargado de recibir imágenes, ejecutar modelos de detección de irregularidades y almacenar resultados georreferenciados. El backend, compuesto por una API REST en FastAPI[2] y un servicio de inteligencia artificial basado en YOLOv5[3] y RabbitMQ[4], constituye el foco exclusivo de esta memoria.

Si bien Geoviality cumple su propósito funcional, su backend presenta una **deuda técnica acumulada** significativa, producto de decisiones de diseño tomadas bajo las fuertes restricciones de tiempo propias de un proyecto académico. Esta deuda se manifiesta en una organización monolítica del código, acoplamiento fuerte entre componentes, manejo limitado de errores y configuraciones de seguridad incompletas. Estos factores degradan la calidad interna del producto, afectando especialmente su **mantenibilidad, fiabilidad y seguridad**. El presente trabajo aborda este problema desde un enfoque de ingeniería de software, utilizando el modelo de calidad ISO/IEC 25010:2023[5] para diagnosticar, planificar y ejecutar mejoras sistemáticas sobre el sistema existente.



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: Diagnóstico y plan de mejora de la calidad de software en un sistema de gestión vial automatizado: Caso Geovality

Nombre del candidato(a): Felipe Ignacio Mellado Olea

Carrera / Grado: Ingeniería Civil Informática

Campus: San Joaquín **Departamento:** Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Marcello 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.

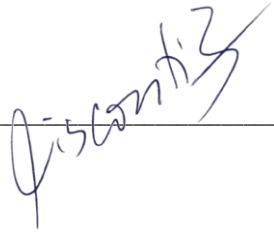
☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 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: 09/10/2026 **Firma:** 

Estudiante o Candidato(a):

Fecha: 09/10/2026 **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.



1.1 Definición del Problema

El deterioro progresivo de las calzadas urbanas —materializado en baches, grietas y hundimientos— constituye un problema crítico para la gestión municipal[6], asociado directamente a un aumento en la siniestralidad vial, daños vehiculares y costos correctivos crecientes[7]. Si bien soluciones tecnológicas emergentes, como **Geoviality**, prometen optimizar este proceso mediante inteligencia artificial y visión por computador, su viabilidad técnica no depende solo de la eficacia de sus algoritmos de detección, sino de la robustez de la ingeniería de software que los soporta.

En la actualidad, el backend de Geoviality enfrenta una crisis de **sostenibilidad técnica**. Construido originalmente bajo la presión de plazos académicos acotados, el sistema priorizó la entrega funcional ("Time-to-Market") por sobre la calidad estructural. Esta estrategia, aunque válida en etapas de prototipado, ha derivado en una **deuda técnica imprudente** que hoy compromete la operación y evolución del sistema. La arquitectura actual presenta un acoplamiento excesivo entre la lógica de negocio, la capa de persistencia y el servicio de inferencia (IA), configurando un "monolito distribuido" donde cualquier modificación trivial conlleva un alto riesgo de introducir regresiones funcionales.

Específicamente, la ausencia de una estrategia de pruebas automatizadas y la carencia de estándares de codificación han generado un código fuente rígido y difícil de mantener. Desde la perspectiva de la **seguridad**, la exposición de credenciales y la falta de validación estricta en los contratos de datos exponen al sistema a vulnerabilidades que serían inaceptables en un entorno productivo real. Por otro lado, en términos de **fiabilidad**, el sistema carece de mecanismos de recuperación ante fallos en el procesamiento asíncrono de imágenes, lo que puede ocasionar pérdida de datos críticos para la gestión vial.

Esta acumulación de deficiencias crea un círculo vicioso: el costo de mantenimiento aumenta exponencialmente mientras que la velocidad de desarrollo disminuye. Sin una intervención profunda, el sistema está condenado a la obsolescencia técnica, impidiendo su transferencia a entornos municipales reales o su escalamiento a grandes volúmenes de datos.

Por consiguiente, el problema de ingeniería que aborda esta memoria no es solo "corregir errores", sino la necesidad imperiosa de **sistematizar el pago de la deuda técnica** acumulada. Se requiere transformar un prototipo académico frágil en un producto de software mantenible, fiable y seguro, mediante la aplicación rigurosa de estándares de calidad (ISO/IEC 25010:2023) y prácticas de ingeniería de software.

1.2 Introducción a la Solución

Para resolver la problemática de deuda técnica identificada, esta memoria propone y ejecuta una estrategia de Mejora de Procesos de Software (SPI) basada en un ciclo iterativo de diagnóstico, planificación, implementación y evaluación.

La solución consiste en el diseño de un **Plan de Mejora Integral** para el backend de Geoviality, del cual se ejecuta una primera iteración completa enfocada en el *hardening* (endurecimiento) de seguridad y la refactorización modular. El proceso metodológico abarca un diagnóstico profundo utilizando herramientas de análisis estático (SonarQube[8], Pylint[9], Bandit[10], Owasp Dependency-Check[11]) y dinámico (OWASP ZAP[12]) combinadas con revisión manual, lo que permite establecer una línea base cuantitativa. Posteriormente, se realiza una planificación priorizada para diseñar una hoja de ruta por fases, seleccionando las intervenciones de mayor impacto y menor costo.

La ejecución técnica contempla la refactorización y *hardening* mediante la reingeniería del código fuente para desacoplar servicios, estandarizar patrones de diseño y corregir vulnerabilidades, siempre cuidando no alterar los contratos funcionales externos. Finalmente, se realiza una verificación de calidad validando las

mejoras mediante la re-ejecución de métricas y un análisis comparativo para demostrar objetivamente la reducción de la deuda técnica según la norma ISO/IEC 25010:2023.

1.3 Objetivos

1.3.1 General

Diagnosticar la deuda técnica del backend de Geoviality, diseñar un plan de mejora evolutivo y ejecutar una primera iteración de refactorización validada, que incremente los índices de mantenibilidad, fiabilidad y seguridad del sistema según el modelo de calidad ISO/IEC 25010:2023.

1.3.2 Específicos

Tabla 1: Objetivos específicos

Fuente: Elaboración propia

Nº	Objetivo específico
1	Caracterizar cuantitativamente la deuda técnica actual de la API y el servicio de IA, generando una línea base de métricas mediante herramientas de análisis estático y de dependencias (SonarQube, Pylint, Bandit, OWASP Dependency-Check), dinámico (OWASP ZAP) y revisión arquitectónica manual.
2	Diseñar un plan de mejora priorizado y dividido en fases incrementales, que defina acciones concretas de refactorización, <i>modularización</i> y seguridad para mitigar los hallazgos del diagnóstico.
3	Implementar la primera iteración del plan de mejora, ejecutando acciones de <i>hardening</i> de seguridad, saneamiento de código (<i>code hygiene</i>) y reorganización modular de la API, validando la integridad funcional de los servicios.
4	Evaluar el impacto de la intervención comparando las métricas de calidad finales contra la línea base inicial, para cuantificar la reducción de deuda técnica y la mejora en los atributos de calidad seleccionados.
5	Sistematizar el proceso de mejora , documentando las decisiones arquitectónicas y metodológicas para facilitar la transferibilidad del conocimiento y futuras iteraciones de mantenimiento en el proyecto.

1.4 Estructura del Documento

Esta memoria se organiza en siete capítulos que documentan el ciclo completo de mejora aplicado. El documento comienza presentando el contexto del proyecto, la problemática de la deuda técnica en Geoviality, la solución propuesta y los objetivos perseguidos. A continuación, el segundo capítulo establece las bases conceptuales sobre gestión vial, el sistema Geoviality, la definición de deuda técnica y el modelo de calidad ISO/IEC 25010:2023 que guía la evaluación. Posteriormente, el tercer capítulo detalla la metodología de evaluación y presenta los hallazgos cuantitativos y cualitativos del estado inicial ("AS-IS") del backend, identificando los puntos críticos a intervenir.

Sobre esta base diagnóstica, el cuarto capítulo propone una estrategia de evolución por fases para el sistema, definiendo la arquitectura objetivo ("TO-BE") y priorizando las acciones para la primera iteración. El quinto capítulo describe técnicamente las actividades de ingeniería realizadas, incluyendo la refactorización de código, reorganización de módulos y aplicación de parches de seguridad. El sexto capítulo expone la evaluación final post-intervención, contrastando las métricas obtenidas con el diagnóstico inicial para validar el éxito de la mejora. Finalmente, el séptimo capítulo sintetiza los logros alcanzados respecto a los objetivos, discute las limitaciones del trabajo y propone líneas de trabajo futuro.



2 Marco Teórico

2.1 Sistemas para la Detección de Irregularidades Viales

En los últimos años, la detección automatizada de irregularidades viales ha avanzado significativamente gracias a la convergencia de la visión por computador, la inteligencia artificial y los sensores móviles[13]. Estos enfoques buscan superar las limitaciones de las inspecciones manuales tradicionales —como las basadas en el índice PCI (Pavement Condition Index)— ofreciendo procesos más rápidos, objetivos y escalables para auditar redes urbanas extensas.

Las soluciones basadas en visión por computador combinan técnicas de procesamiento de imágenes con redes neuronales convolucionales (CNN) profundas[14], capaces de detectar y clasificar daños superficiales (grietas, baches, desprendimientos) a partir de imágenes capturadas desde vehículos en movimiento o dispositivos móviles. Paralelamente, los métodos apoyados en sensores inerciales utilizan acelerómetros y giroscopios para registrar las vibraciones asociadas a las irregularidades del pavimento[15], [16], o tecnologías más costosas basadas en LiDAR[17]. Si bien los sensores son económicos y escalables, carecen de la precisión espacial y la capacidad de clasificación semántica que ofrecen los sistemas basados en visión.

En este contexto, las arquitecturas híbridas que combinan la detección visual con la georreferenciación precisa se han posicionado como el estándar moderno para la gestión de infraestructura inteligente (*Smart Infrastructure*), permitiendo no solo la identificación del daño, sino su mapeo y trazabilidad en el tiempo.

2.2 Geoviality

Geoviality es un sistema de gestión vial inteligente cuyo propósito principal es detectar, clasificar y mapear irregularidades en la calzada —como baches, grietas o desniveles— mediante técnicas de visión por computador e inteligencia artificial. Se enmarca en soluciones de *Smart Mobility* y *Smart Infrastructure* orientadas a mejorar los procesos de mantenimiento y planificación vial mediante automatización, *sensorización* y análisis de datos[1].

El sistema permite operar tanto en modo peatón (captura manual) como en modo vehicular (captura automática durante la conducción). En la etapa de detección, utiliza el modelo YOLOv5[3], entrenado en PyTorch[18], para identificar irregularidades en muy poco tiempo, directamente desde la cámara del teléfono. Este enfoque busca un equilibrio entre precisión técnica, rendimiento operativo y costos de implementación, facilitando su adopción por municipalidades u organizaciones con recursos acotados.

Geoviality está compuesto por dos capas frontales —una aplicación móvil y una plataforma web— y un backend que constituye el foco de esta memoria. El backend se organiza en dos servicios principales: una API REST desarrollada en FastAPI[2], responsable de la gestión de usuarios, recepción y validación de datos y comunicación con la base de datos; y un servicio de detección con IA, que procesa las imágenes mediante YOLOv5[3]. Ambos servicios se comunican a través de una cola de mensajería RabbitMQ[4], lo que permite desacoplar el procesamiento intensivo de la IA del flujo de solicitudes de la API y soportar un esquema asíncrono de intercambio de mensajes.

Los datos estructurados se almacenan en una base de datos NoSQL MongoDB[19], mientras que las imágenes se conservan en el sistema de archivos del servidor de la API. Esta configuración híbrida combina la flexibilidad documental para metadatos geospaciales con la simplicidad del almacenamiento de archivos. La solución se ejecuta mediante contenedores Docker[20], lo que facilita su despliegue y la futura evolución hacia una arquitectura de servicios más distribuida.

2.2.1 Arquitectura general del sistema

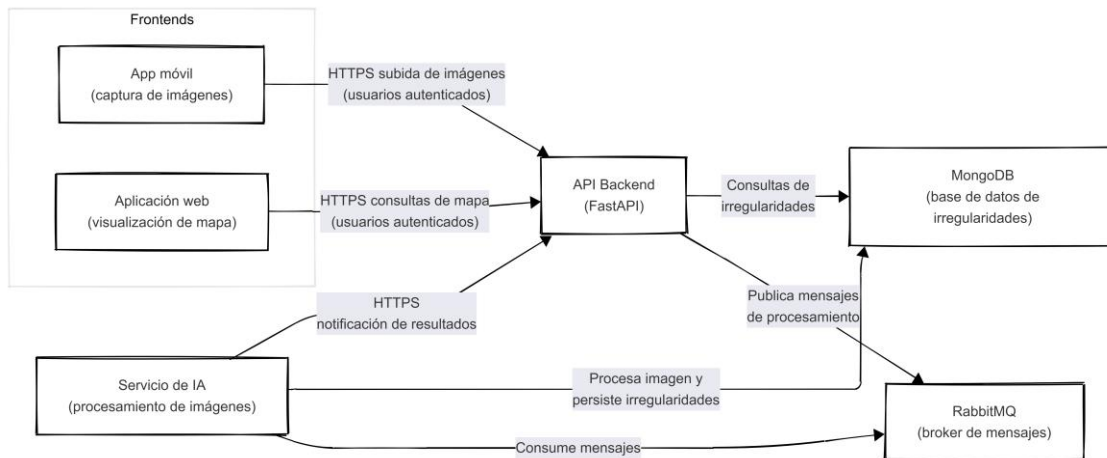


Ilustración 1: Diagrama del sistema

Fuente: Elaboración propia

Los componentes principales del sistema son:

- **Frontends:** Una aplicación móvil optimizada para la captura en movimiento y una plataforma web para la visualización administrativa y gestión de cuadrillas.
- **API Backend:** Desarrollada en **FastAPI**[2], actúa como el orquestador central del sistema. Es responsable de la autenticación de usuarios, la validación de esquemas de entrada y la exposición de endpoints REST para la consulta de datos geoespaciales.
- **Servicio de IA (Worker):** Un **componente de procesamiento asíncrono** encargado de la inferencia computacional. Recibe referencias a imágenes, ejecuta el modelo de detección y extrae metadatos (tipo de daño, confianza, coordenadas) sin bloquear el hilo principal de la API.
- **Infraestructura de Mensajería:** Se utiliza **RabbitMQ**[4] como broker de mensajes para gestionar la comunicación asíncrona entre la API y el servicio de IA. Este componente es crítico para soportar la carga variable generada por el "modo vehicular", actuando como buffer para evitar la saturación del servidor.
- **Persistencia Híbrida:** Se emplea **MongoDB**[19] para el almacenamiento de metadatos y consultas geoespaciales (GeoJSON), mientras que las evidencias fotográficas se almacenan en el sistema de archivos local del servidor.

2.3 Deuda Técnica

En la ingeniería de software, la "deuda técnica" es una metáfora introducida originalmente por Ward Cunningham[21], quien comparó la complejidad técnica acumulada por decisiones de diseño apresuradas o subóptimas con una deuda financiera. Si bien asumir cierta deuda puede ser estratégico para agilizar la entrega inicial ("Time-to-Market"), si esta no se "paga" mediante refactorización oportuna, acumula "intereses". Estos intereses se manifiestan como un aumento progresivo en el esfuerzo de mantenimiento, mayor tasa de errores y una dificultad creciente para evolucionar el sistema.

En el contexto de Geovality, la deuda técnica identificada se clasifica en las siguientes dimensiones críticas:

- **Deuda de Código:** Presencia de complejidad ciclomática excesiva, duplicación de lógica y violaciones de estándares de estilo que dificultan la lectura del código.



- **Deuda Arquitectónica:** Acoplamiento fuerte entre componentes que deberían ser independientes (ej. lógica de negocio mezclada con acceso a datos), configurando una estructura cercana a un monolito no modular ("Big Ball of Mud").
- **Deuda de Pruebas:** Ausencia o insuficiencia de pruebas automatizadas (unitarias y de integración), lo que incrementa significativamente el riesgo de regresiones ante cualquier cambio.
- **Deuda de Seguridad:** Configuraciones por defecto inseguras, gestión inadecuada de secretos y dependencias desactualizadas.
- **Deuda de Documentación:** Carencia de documentación técnica actualizada y diagramas de arquitectura vigentes, así como la falta de contratos de API explícitos. Esta deuda dificulta el onboarding de nuevos desarrolladores y oscurece la comprensión del funcionamiento interno del sistema.

2.4 Calidad de Software y Modelo ISO/IEC 25010:2023

La calidad del software se define como el grado en que un producto satisface las necesidades explícitas e implícitas de sus partes interesadas[22]. Para evaluar y objetivar esta calidad, este trabajo se rige por la norma internacional **ISO/IEC 25010:2023**[5], la cual actualiza el modelo de calidad del producto (*Product Quality Model*). Esta norma define un modelo compuesto por nueve características principales: Adecuación Funcional (*Functional Suitability*), Eficiencia de Desempeño (*Performance Efficiency*), Compatibilidad (*Compatibility*), Capacidad de Interacción (*Interaction Capability*), Fiabilidad (*Reliability*), Seguridad (*Security*), Mantenibilidad (*Maintainability*), Flexibilidad (*Flexibility*) y Seguridad de Funcionamiento (*Safety*).

Dado el alcance de la refactorización del backend abordada en esta memoria, el diagnóstico y el plan de mejora priorizan tres de estas características:

1. **Mantenibilidad (Maintainability):** Representa la capacidad del sistema para ser modificado de manera efectiva y eficiente. Se evalúa a través de subcaracterísticas como la **modularidad** (cambios localizados), la **analizabilidad** (facilidad de diagnóstico) y la **modificabilidad**.
2. **Fiabilidad (Reliability):** Capacidad del sistema para desempeñar sus funciones especificadas sin fallos bajo condiciones determinadas. Incluye la **tolerancia a fallos** (operación continua ante errores) y la **capacidad de recuperación** (restablecimiento de datos).
3. **Seguridad (Security):** Grado en que el sistema protege la información y los datos, asegurando la **confidencialidad, integridad y autenticidad** frente a accesos no autorizados.

2.5 Fundamentos de Arquitectura y Modularidad

La arquitectura de software no solo define la estructura del sistema, sino que determina sus características de calidad fundamentales. Según Richards y Ford[23], la **modularidad** es el principio organizativo clave para gestionar la complejidad en sistemas crecientes. Su efectividad se evalúa principalmente a través de dos métricas estructurales:

- **Cohesión:** Medida en que las partes de un módulo pertenecen juntas funcionalmente. Una alta cohesión es deseable, ya que indica que el módulo tiene una responsabilidad única y clara, facilitando su comprensión y mantenimiento.
- **Acoplamiento:** Grado de interdependencia entre módulos. Un bajo acoplamiento es crítico para permitir la evolución independiente de los servicios. En arquitecturas monolíticas no estructuradas, un alto acoplamiento (aférente y eférente) genera rigidez, donde cambios menores provocan efectos en cascada en todo el sistema, dificultando el despliegue y las pruebas.

2.6 Estrategias de Modernización y Refactorización

La modernización de sistemas heredados (*legacy*) o con alta deuda técnica conlleva riesgos significativos si se opta por una reescritura total (Big Bang). La ingeniería de software favorece enfoques incrementales que permitan mejorar el sistema sin detener su operación.

2.6.1 Patrón Strangler Fig

Para la evolución arquitectónica, se utiliza el patrón **Strangler Fig** (*Higuera Estranguladora*), sistematizado por Sam Newman[24]. Este patrón propone una estrategia de migración gradual donde las nuevas funcionalidades o las refactorizaciones de módulos existentes se construyen alrededor del sistema legado. Estas nuevas implementaciones interceptan las llamadas y reemplazan progresivamente a los componentes antiguos hasta que el sistema original puede ser retirado ("*estrangulado*") con seguridad. Esta aproximación reduce el riesgo de regresiones catastróficas y permite entregar valor continuo. Para sistemas monolíticos, la literatura recomienda una fase preliminar de **Logical Decomposition** (Descomposición Lógica), creando fronteras modulares internas antes de la separación física.

2.6.2 Higiene de Código

A nivel de implementación, se aplican prácticas de higiene de código inspiradas en la "Regla del Boy Scout": *dejar el código más limpio de lo que se encontró*. Esto implica realizar micro-refactorizaciones continuas — como mejorar nombres de variables, simplificar condicionales o extraer métodos— durante el ciclo de desarrollo normal. Estas acciones, aunque pequeñas individualmente, reducen la entropía del software y mejoran la mantenibilidad a largo plazo.

2.7 Seguridad Aplicada en el Ciclo de Vida

La seguridad no debe ser una característica añadida al final del desarrollo, sino integrada en todo el ciclo de vida, enfoque conocido como Shift-Left Security. Esto implica adoptar prácticas de desarrollo seguro que aborden las vulnerabilidades más comunes, como las descritas en el **OWASP Top 10**[25].

Para APIs REST modernas, los controles críticos incluyen:

- **Gestión de Identidad y Acceso:** Implementación de estándares robustos como **OAuth2** y **JWT** (JSON Web Tokens) para autenticación y autorización.
- **Protección de la Capa de Transporte:** Uso forzoso de HTTPS/TLS y cabeceras de seguridad HTTP (como HSTS y CSP) para proteger los datos en tránsito.
- **Gestión de Secretos:** Almacenamiento seguro de credenciales y claves de API fuera del código fuente, utilizando variables de entorno o gestores de secretos.

2.8 Estrategias de Pruebas y Verificación de Software

El proceso de pruebas (testing) constituye un componente esencial en el aseguramiento de la calidad del software (SQA). Desde una perspectiva técnica, esta actividad no se concibe como un medio para demostrar la funcionalidad del software, sino como el proceso de ejecutar un programa con la intención deliberada de identificar fallos[22]. Bajo este enfoque, se considera que un caso de prueba es exitoso únicamente cuando logra revelar un defecto no detectado con anterioridad.

En el contexto de la modernización de sistemas legados y la reducción de deuda técnica, las pruebas cumplen un rol doble: validar que los nuevos componentes cumplan los requisitos (verificación) y asegurar que los cambios introducidos no rompan funcionalidades existentes (prevención de regresiones).

2.8.1 Niveles de Prueba y la Pirámide de Testing

Para una estrategia de pruebas eficiente, la ingeniería de software organiza las pruebas en niveles de granularidad:

- **Pruebas Unitarias (Unit Testing):** Se centran en la verificación de la unidad más pequeña de diseño de software (componente o módulo). Utilizan un enfoque de "caja blanca" (white-box testing) para



asegurar que las rutas lógicas internas y los flujos de control operen correctamente. Son la base de la pirámide de pruebas por su bajo costo y alta velocidad de ejecución.

- **Pruebas de Integración:** Validan la interacción entre módulos que han sido probados unitariamente. Su objetivo es detectar errores en las interfaces y en el flujo de datos entre componentes, aspecto crítico en la arquitectura de Geoviality dada la comunicación entre la API y el servicio de IA.
- **Pruebas de Sistema:** Verifican que el sistema completo e integrado cumpla con los requisitos especificados, operando usualmente bajo un enfoque de "caja negra" (black-box testing) donde se evalúa la funcionalidad observable sin considerar la estructura interna.

2.8.2 Pruebas de Regresión y Refactorización

La refactorización —el proceso de mejorar la estructura interna del software sin alterar su comportamiento externo— conlleva un riesgo inherente de introducir nuevos defectos. Para mitigar este riesgo, es imprescindible aplicar **pruebas de regresión**.

Las pruebas de regresión se definen como la re-ejecución de un subconjunto de pruebas preexistentes, cuyo objetivo es garantizar que las modificaciones introducidas en el código —tales como correcciones, mejoras o adaptaciones— no propaguen efectos colaterales adversos[22]. En sistemas que presentan una elevada deuda técnica y una baja cobertura de pruebas automatizadas, la refactorización implica un alto riesgo, lo cual fundamenta la priorización de la 'cobertura de código' (code coverage) como métrica clave de calidad dentro de los planes de mejora.

2.9 Selección y Justificación de Herramientas de Análisis (SQA)

Para objetivar el diagnóstico de la deuda técnica y validar las mejoras, se seleccionó un conjunto de herramientas de análisis estático (SAST) y dinámico (DAST), justificadas por su capacidad de integración con el ecosistema Python y su adopción industrial:

- **SonarQube:** Plataforma central de calidad seleccionada por su capacidad para agregar métricas heterogéneas (bugs, vulnerabilidades, code smells, duplicación) y proporcionar una estimación temporal de la deuda técnica (Technical Debt Ratio)[8].
- **Pylint:** Herramienta de análisis estático para Python, elegida por su análisis semántico profundo capaz de detectar errores lógicos y validar el cumplimiento del estándar PEP-8[9], [26].
- **Emerge-viz:** Utilizada para la visualización estructural del código, permitiendo identificar gráficamente clústeres de acoplamiento y dependencias cíclicas[27].
- **OWASP ZAP (Zed Attack Proxy):** Estándar *de facto* para pruebas de seguridad dinámicas. Se utiliza para validar la postura de seguridad de la API en ejecución, detectando malas configuraciones no visibles en el código estático[12].
- **Bandit:** Analizador estático de seguridad específico para Python, diseñado para encontrar patrones de codificación inseguros comunes en este lenguaje[10].
- **OWASP Dependency-Check:** Herramienta de Análisis de Composición de Software (SCA) fundamental para detectar vulnerabilidades conocidas (CVEs) en las librerías y dependencias de terceros utilizadas en el proyecto, mitigando riesgos de seguridad en la cadena de suministro de software[11].

3 Diagnóstico

3.1 Metodología de evaluación

El diagnóstico se centra en el backend de Geoviality (API y servicio de IA) y tiene por objetivo caracterizar la deuda técnica existente en relación con las características de calidad priorizadas en el Marco Teórico: **Mantenibilidad, Fiabilidad y Seguridad**.

Para objetivar la evaluación, se adoptó un enfoque híbrido que combina:

1. **Análisis Estático y de Composición (SAST/SCA):** Ejecución de SonarQube, Pylint, Bandit y OWASP Dependency-Check para medir la deuda de código, dependencias vulnerables y seguridad sin ejecutar el software.
2. **Análisis Estructural:** Uso de Emerge-viz para visualizar el acoplamiento y la cohesión arquitectónica.
3. **Análisis Dinámico (DAST):** Escaneo con OWASP ZAP sobre la API en ejecución para detectar vulnerabilidades de configuración.
4. **Revisión Manual:** Inspección cualitativa de contratos de API, manejo de errores y lógica de negocio.

Los resultados obtenidos constituyen la **línea base ("AS-IS")** frente a la cual se medirán las mejoras implementadas en la fase de resultados.

3.2 Análisis cuantitativo con herramientas

Esta sección presenta los hallazgos métricos, interpretados bajo los estándares de ingeniería definidos previamente.

3.2.1 SonarQube: Mantenibilidad y Complejidad

El análisis de SonarQube revela indicadores críticos sobre la estructura interna del código, tal como se detalla a continuación.

Tabla 2: Métricas obtenidas de SonarQube

Fuente: Elaboración propia

Métrica	Resultado	Interpretación Técnica
Security Rating	A (0 issues)	No se detectan inyecciones obvias en análisis estático, aunque esto no garantiza la seguridad en tiempo de ejecución.
Reliability Rating	E (4 issues)	Crítico. La calificación 'E' indica presencia de bugs que comprometen la tolerancia a fallos. Requiere corrección inmediata.
Maintainability	A (49 code smells)	Engañoso. Aunque el rating es alto, la presencia de 49 code smells dispersos sugiere una degradación progresiva de la higiene del código.
Technical Debt / Debt Ratio	4h 58min / 0.5%	Esfuerzo de remediación de code smells. El bajo ratio es nominal; no refleja la complejidad del acoplamiento ni el riesgo por falta de pruebas automatizadas.
Code Coverage	0.0%	Hallazgo Crítico (Deuda de Pruebas). La ausencia total de pruebas impide realizar pruebas de regresión, elevando el riesgo de cualquier refactorización.
Cyclomatic Complexity	180 (Total)	Alta acumulación. Múltiples métodos individuales superan el umbral de riesgo recomendado ($v(G) > 10$), indicando dificultad de prueba .
Cognitive Complexity	173	Nivel elevado de anidamiento lógico (<i>nested if/loops</i>), lo que dificulta la analizabilidad por parte de nuevos desarrolladores.



Si bien el Security Rating parece favorable, la métrica de **complejidad ciclomática** agregada (180) y la **cobertura de código** (0.0%) configuran un escenario de alto riesgo. La complejidad cognitiva elevada correlaciona directamente con la dificultad de mantenimiento descrita como deuda técnica. Específicamente, la falta de una red de seguridad de pruebas automatizadas clasifica esta deuda como "imprudente" en el contexto de una operación productiva, ya que no existen garantías de estabilidad ante cambios, y los métodos que exceden los umbrales estándar de complejidad presentan una alta probabilidad de contener defectos latentes.

Esta brecha entre el bajo Technical Debt Ratio (0.5%) y la fragilidad observada en el sistema se explica por la naturaleza de la métrica de SonarQube, que prioriza la mantenibilidad a nivel de micro-refactorización. Sin embargo, al cruzar estos datos con la **cobertura de código del 0.0%** y la **complejidad cognitiva (173)**, se concluye que el sistema posee una 'deuda invisible' significativa. Bajo el modelo ISO/IEC 25010:2023, esto implica que, aunque el código individual sea 'limpio', la **analizabilidad** y **modificabilidad** globales están severamente comprometidas por la falta de una red de seguridad (pruebas) y una estructura de control excesivamente anidada.

3.2.2 Pylint: Calidad de Código y Estándares

Tabla 3: Métricas obtenidas de Pylint

Fuente: Elaboración propia

Métrica	Resultado	Interpretación Técnica
Puntaje Global (Ambos)	5.56 / 10	Cumplimiento parcial de normas PEP-8. Requiere mejoras en estilo, documentación y consistencia de nombres.

El puntaje global de ambos (tanto API como IA) **5.56 sobre 10** evidencia una desviación significativa respecto al estándar PEP-8. Esta baja puntuación trasciende lo estético e indica inconsistencias en la nomenclatura, importaciones cíclicas potenciales y falta de docstrings. Esto impacta negativamente en la **analizabilidad** del sistema, dificultando que nuevos desarrolladores comprendan el propósito de los módulos sin tener que leer la implementación línea por línea.

3.3 Emerge-viz: Arquitectura y Modularidad

Tabla 4: Métricas obtenidas de Emerge-viz

Fuente: Elaboración propia

Métrica	Resultado IA	Resultado API	Interpretación Técnica
Métodos por archivo	3.71	9.25	La API muestra baja granularidad, concentrando demasiadas responsabilidades en pocos archivos.
Promedio SLOC	83	146	Archivos extensos en la API, dificultando la lectura y el control de versiones.
Fan-in Máximo	5	5	Reutilización moderada, dentro de rangos aceptables.
Fan-out Máximo	8	20	Crítico (Alto Acoplamiento). Un módulo depende de otros 20, creando fragilidad arquitectónica.

Existe una clara asimetría estructural. Mientras el servicio de IA presenta una cohesión aceptable con módulos granulares, la API exhibe características de un sistema monolítico no estructurado. El alto Fan-out (20) confirma un acoplamiento excesivo entre la lógica de rutas, controladores y acceso a datos. Este patrón de "Big Ball of Mud" impide la evolución independiente de los componentes y complica el aislamiento de fallos, justificando la necesidad de una reorganización arquitectónica orientada a dominios.

3.4 Seguridad (Bandit, OWASP ZAP y Dependency Check)

El diagnóstico de seguridad integra la visión estática, dinámica y de dependencias para evaluar la postura de seguridad global.

I. Bandit

Tabla 5: Métricas obtenidas de Bandit

Fuente: Elaboración propia

Módulo	Regla (ID)	Severidad	Ubicación	Hallazgo
API	B106	Baja	app/routes.py:221	Uso de literales <i>hardcodeados</i> asociados a esquemas de autenticación.
IA	B113	Media	funcs.py:131	Llamada HTTP sin timeout definido, susceptible a bloqueos indefinidos.

Aunque los hallazgos son puntuales, el problema de llamadas HTTP sin timeout en el servicio de IA representa un riesgo de disponibilidad. En un entorno de microservicios, una dependencia externa lenta podría agotar los hilos del consumidor, deteniendo el procesamiento de imágenes. Asimismo, la presencia de credenciales o literales en el código viola el principio de separación entre configuración y código.

II. OWASP ZAP

Tabla 6: Hallazgos mediante OWASP ZAP

Fuente: Elaboración propia

Hallazgo	Severidad	Instancias	Impacto Técnico
Política CSP incompleta	Media	1	Permite la carga de recursos no controlados, ampliando la superficie de XSS.
Script/Style 'unsafe-inline'	Media	2	Facilita la ejecución de scripts maliciosos inyectados en el DOM.
CORS permisivo (*)	Media	1	Permite que cualquier origen consuma la API, riesgo de abuso de credenciales.
Falta cabecera HSTS	Media	1	Permite downgrade a HTTP, exponiendo el tráfico a interceptación.
X-Frame-Options ausente	Baja	1	Riesgo de <i>Clickjacking</i> al permitir embeber el sitio en <i>iframes</i> .
Falta SRI / Cache débil	Baja	2	Riesgo de modificación de scripts externos y almacenamiento de datos sensibles en caché.

El escaneo dinámico revela una falta generalizada de *hardening* en la capa de transporte y presentación. La configuración laxa de CORS y la ausencia de cabeceras de seguridad (HSTS, CSP) exponen la API a vectores de ataque comunes como XSS y *Clickjacking*. Esto evidencia que la seguridad no fue integrada en el ciclo de desarrollo temprano (Shift-Left), dejándola como una consideración tardía.

III. OWASP Dependency Check

Tabla 7: Métricas obtenidas de OWASP Dependency Check

Fuente: Elaboración propia

Proyecto	Vulnerabilidades	Severidad	Detalle
IA	1	Crítica	Vulnerabilidad conocida en la versión utilizada de la librería PyTorch.
API	0	-	Sin vulnerabilidades conocidas reportadas.

La identificación de una vulnerabilidad crítica en PyTorch constituye una **deuda de seguridad** externa significativa. Al ser el núcleo del motor de inferencia, mantener esta dependencia desactualizada expone al



sistema a fallos de seguridad ya documentados y explotables, lo que requiere una actualización prioritaria de las librerías base.

3.5 Revisión Manual de código y Arquitectura

Dado que las herramientas automáticas no pueden evaluar la semántica del negocio ni la corrección lógica completa, se realizó una inspección manual para identificar deuda arquitectónica y riesgos semánticos no detectables automáticamente (ver el detalle completo de hallazgos en el **Anexo A**). Los hallazgos más relevantes se resumen a continuación.

Tabla 8: Revisión Manual de código

Fuente: Elaboración propia

ID	Módulo	Hallazgo crítico	Severidad	Tipo de deuda técnica	Categoría / principio afectado	Impacto principal
1	api/app/routes.py	Endpoint /data/processed_image expuesto sin autenticación ni validación estricta de entrada y salida.	Alta	Seguridad	OWASP – autenticación / autorización; validación de entrada	Incrementa el riesgo de inserción de datos no confiables y uso malicioso del endpoint.
2	api/app/controllers.py	Lógica invertida en operaciones CRUD: se interpreta éxito como error y viceversa.	Alta	Código	Contratos REST/BD; corrección funcional	Genera inconsistencias silenciosas en la persistencia y en las respuestas hacia los clientes.
3	api/app/models.py	Contraseñas en modelos de usuario almacenadas como texto plano, sin hash ni mecanismo de protección asociado.	Alta	Seguridad	OWASP – almacenamiento de contraseñas	Compromete directamente credenciales ante una fuga o acceso indebido a la base de datos.
4	ia/main.py	Callback de consumo de cola sin manejo de excepciones previo a basic_ack.	Alta	Arquitectura	Fiabilidad de Mensajería; manejo robusto de errores	Puede provocar la pérdida de mensajes o congelar el consumidor, interrumpiendo el servicio.
5	ia/ia_predictor.py	Uso de exit() ante archivo no válido; derriba el proceso completo de IA.	Alta	Arquitectura	Disponibilidad; manejo de errores	Provoca interrupción abrupta del servicio de IA y pérdida de mensajes en curso sin reintento.
6	ia/funcs.py	Conversión directa de resultados de base de datos a GeoJSON sin validar None.	Alta	Código	Null-safety; programación defensiva	Genera excepciones en el flujo IA→API, interrumpiendo el procesamiento de irregularidades.

7	ia/detector_ia.py	Construcción de rutas con \ y os.getcwd(); no es portable a Linux ni a entornos basados en contenedores.	Alta	Arquitectura	Portabilidad; configuración externa	Puede provocar fallas completas del pipeline de IA en despliegues Linux/container.
8	api/app/controllers.py	SECRET_KEY leído sin validación; puede ser None y romper la generación / validación de JWT.	Alta	Seguridad	Gestión de secretos; integridad de sesión	Conduce a tokens inválidos y debilitamiento del esquema de autenticación.
9	api/app/routes.py	WebSocket depende de event_queue inexistente; provoca excepción y cierre abrupto del canal.	Alta	Arquitectura	Resiliencia; robustez de dependencias	Impide emitir eventos en tiempo real y genera fallas recurrentes en el canal de comunicación.
10	ia/ia_predictor.py	Cálculo de confianza multiplica por 100 pero se compara con un umbral 0-1; condición de filtrado incorrecta.	Media	Código	Consistencia de métricas; validación de datos	Produce falsos positivos / negativos en detecciones, afectando la calidad de la información.

Los puntos más críticos detectados fueron:

1. **Manejo de Errores en RabbitMQ:** El callback de consumo de mensajes no implementa bloques try/except robustos antes de enviar la confirmación (ack). Un error de procesamiento provoca la pérdida del mensaje o el bloqueo del consumidor, violando la tolerancia a fallos.
2. **Inconsistencia de Contratos:** La API expone endpoints que retornan estructuras de datos no *tipadas* estrictamente, lo que genera errores de integración silenciosos con el frontend.
3. **Gestión de Secretos:** Se verificó la lectura de variables críticas como SECRET_KEY sin validación de nulidad, comprometiendo la integridad de la autenticación.

3.5.1 Hallazgos de Documentación

Se evaluó el estado de la documentación técnica disponible frente a las necesidades de mantenimiento del sistema, identificando una carencia estructural severa:

- **Documentación de API (Nivel 1):** Existe documentación automática generada por **FastAPI** (Swagger UI / OpenAPI). Si bien esto permite probar endpoints individuales, **carece de descripciones de negocio**, ejemplos de uso y explicaciones sobre los códigos de error, limitando su utilidad para integradores externos.
- **Ausencia de Documentación Arquitectónica:** No existen diagramas actualizados que expliquen la interacción entre componentes, la infraestructura de despliegue o el flujo de datos asíncrono. El conocimiento sobre la arquitectura reside tácitamente en el desarrollador original, creando un riesgo crítico de "Factor de Bus".
- **Falta de Registro de Decisiones:** No hay registros de decisiones de arquitectura que expliquen por qué se eligieron ciertas tecnologías o patrones, dificultando que futuros mantenedores entiendan el contexto de las decisiones pasadas.

Esta "caja negra" documental eleva drásticamente la curva de aprendizaje y el riesgo de introducir defectos por incompreensión del sistema.



3.6 Síntesis del Diagnóstico

La triangulación de métricas automáticas y revisión manual permite caracterizar la deuda técnica de Geoviality en **cinco dimensiones clave**, confirmando la gravedad de la situación actual:

1. **Deuda de Pruebas (Crítica):** Con una cobertura del **0.0%**, el sistema carece de capacidad de verificación automática. Según los principios de ingeniería de software, esto sitúa al proyecto en un escenario de **alto riesgo de regresión**, donde cualquier refactorización necesaria para pagar otras deudas podría introducir nuevos fallos indetectables.
2. **Deuda Arquitectónica (Alta):** La API presenta un acoplamiento eferente excesivo (Fan-out 20) y baja cohesión, configurando una estructura monolítica difícil de mantener y escalar modularmente.
3. **Deuda de Documentación (Alta):** La dependencia exclusiva de la documentación autogenerada (Swagger) es insuficiente. La falta de diagramas arquitectónicos y manuales de operación impide la transferencia efectiva de conocimiento y la mantenibilidad a largo plazo.
4. **Deuda de Fiabilidad (Media):** La falta de manejo defensivo de excepciones en procesos asíncronos y llamadas externas amenaza la continuidad operativa y la tolerancia a fallos del sistema.
5. **Deuda de Seguridad (Media):** La presencia de vulnerabilidades en dependencias críticas y la falta de *hardening* en la configuración HTTP exponen al sistema a riesgos evitables.

En conjunto, estas dimensiones revelan que el esfuerzo de remediación estimado por las herramientas automáticas es solo la base de un problema mayor: la verdadera deuda técnica en Geoviality incluye un 'interés invisible' representado por el alto riesgo de regresión estructural. Sin una red de pruebas y con la opacidad documental detectada, cualquier intervención necesaria para pagar las deudas de código o seguridad pone en peligro la integridad funcional de todo el sistema.

En conclusión, el backend de Geoviality es funcional pero **frágil y opaco**. La deuda técnica acumulada impide su evolución segura. Por tanto, el Plan de Mejora debe priorizar tres frentes simultáneos: la **refactorización arquitectónica**, la **creación de una red de seguridad** (pruebas y *hardening*) y la **documentación explícita** de la solución para garantizar su sostenibilidad.

4 Plan de mejora

El plan de mejora define la hoja de ruta para transformar el backend de Geoviality desde su estado actual — caracterizado por una alta deuda técnica y fragilidad arquitectónica— hacia una solución robusta, mantenible y segura. Este capítulo detalla la estrategia de solución, la arquitectura objetivo (TO-BE) y la planificación por fases, priorizando las intervenciones según su impacto en los atributos de calidad del modelo ISO/IEC 25010:2023.

4.1 Estrategia de Solución

Para abordar la problemática descrita en el diagnóstico, se evaluaron distintas estrategias de intervención bajo criterios de costo, riesgo y continuidad operativa.

4.1.1 Evaluación de Alternativas

Se analizó la posibilidad de una reescritura completa del sistema (Rewrite o Greenfield). Si bien esta opción permitiría iniciar con una base limpia, fue descartada debido a su alto costo ("The Big Rewrite"), el riesgo de perder lógica de negocio implícita ya validada y la imposibilidad de entregar valor incremental a corto plazo.

En su lugar, se optó por una estrategia de **refactorización evolutiva**. Este enfoque permite "pagar" la deuda técnica progresivamente, manteniendo el sistema funcional en todo momento. La justificación técnica reside en que el núcleo algorítmico (detección YOLOv5) es sólido; el problema radica en la estructura que lo envuelve, la cual es corregible mediante ingeniería de software disciplinada.

4.1.2 Enfoque Metodológico

La ejecución del plan se rige por tres principios rectores:

1. **Patrón Strangler Fig (Descomposición Lógica):** Se ejecutará la primera etapa del patrón, enfocada en el desacoplamiento lógico de los dominios dentro del repositorio existente. Esto permite sanear la deuda técnica y definir contratos estrictos (Bounded Contexts) sin introducir prematuramente la complejidad de una arquitectura distribuida.
2. **Mejora Continua (PDCA):** Cada fase sigue un ciclo de Planificar, Hacer, Verificar y Actuar, asegurando que cada incremento reduzca métricas de deuda técnica antes de avanzar.
3. **Documentación Viva (Living Documentation):** Para mitigar la alta deuda de documentación detectada, se rechaza el enfoque de "documentar al final". La documentación técnica (Swaggers) se generará y actualizará **simultáneamente** con el código en cada fase, garantizando que el sistema deje de ser una "caja negra" desde la primera iteración.

4.2 Arquitectura Objetivo (TO-BE)

La visión arquitectónica apunta a disolver el "monolito distribuido" actual en favor de una **arquitectura orientada a servicios por dominio**, gobernada por contratos explícitos y alta cohesión.

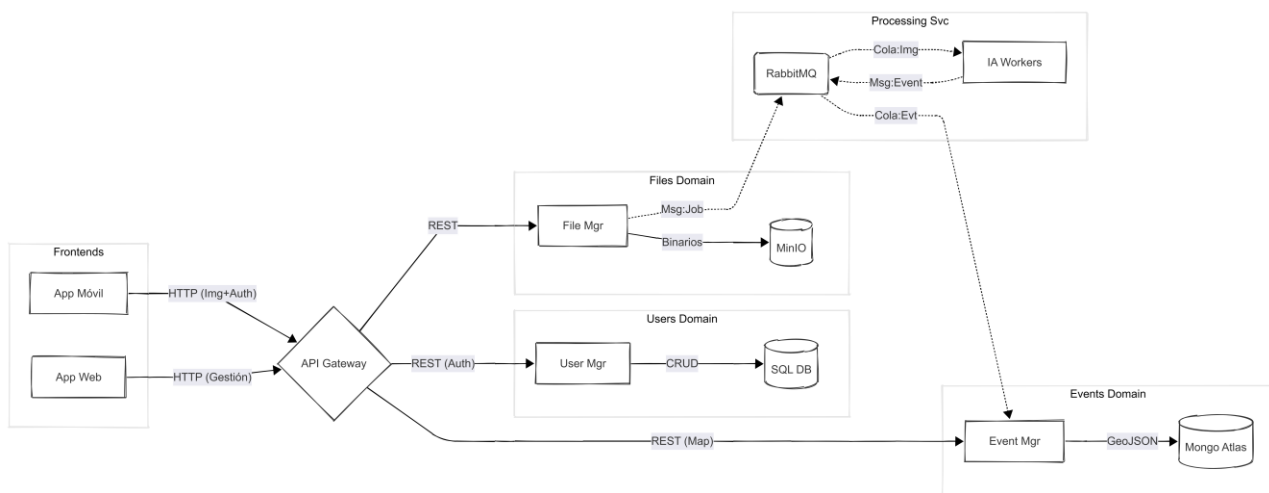


Ilustración 2: Diagrama de Arquitectura Objetivo (TO-BE) de Geoviality

Fuente: Elaboración propia

Como se muestra en la **Ilustración 2**, el diseño propuesto introduce un **API Gateway** como punto único de entrada, desacoplando a los clientes móviles y web de la complejidad interna del backend.

Los componentes esenciales de esta arquitectura son:

- **API Gateway:** Centraliza preocupaciones transversales como la autenticación, la limitación de tasa (rate limiting) y el enrutamiento inteligente de solicitudes. Esto protege a los servicios internos y simplifica la interfaz pública.
- **Servicios de Dominio:** La lógica de negocio se segrega en módulos autónomos con propiedad de datos clara, eliminando el acoplamiento por base de datos compartida:
 - *Gestión de Usuarios:* Responsable exclusivo de credenciales y perfiles.
 - *Gestión Vial (Calles/Veredas):* Encargado de la lógica geoespacial.
 - *Gestión de Archivos:* Abstracción para el almacenamiento de evidencias (soportando S3/MinIO), eliminando la dependencia rígida del sistema de archivos local.



- **Bus de Eventos (RabbitMQ):** Se mantiene y fortalece como la columna vertebral para la comunicación asíncrona. Se formalizan los esquemas de mensajes (eventos de dominio) para garantizar la interoperabilidad robusta entre el servicio de IA y los gestores de datos, asegurando que el procesamiento intensivo no bloquee la experiencia del usuario.

Esta arquitectura busca maximizar la **mantenibilidad** (al reducir el impacto de los cambios) y la **fiabilidad** (al aislar fallos en servicios específicos), alineándose con los objetivos de calidad del modelo ISO/IEC 25010:2023.

4.3 Plan de Mejora por Fases

La transición hacia la arquitectura TO-BE se organiza en fases incrementales. Este trabajo abarca la ejecución de las Fases 0, 1 y 2 (Primera Iteración). Es importante destacar que la **deuda de documentación** se aborda de manera transversal en estas fases iniciales, y no solo al final.

Tabla 9: Plan de mejora

Fuente: Elaboración propia

Fase	Objetivo	Acciones Clave
0. Hardening Mínimo	Mitigar riesgos críticos de seguridad y estabilidad inmediata.	Se aplicará el saneamiento de la configuración CORS y cabeceras HTTP, implementación de hashing seguro de contraseñas, validación estricta de secretos (SECRET_KEY) al arranque y manejo explícito de confirmaciones (Ack) en RabbitMQ para evitar la pérdida de mensajes.
1. Reorganización Modular y Documental	Romper el monolito lógico y pagar deuda de documentación API.	Implementación de la fase de Estrangulamiento Lógico: Reestructuración de la API en dominios aislados (Usuarios, Vialidad) dentro del mismo monolito. Se definen fronteras estrictas que impiden el acceso cruzado a datos, simulando una separación física. Acción Documental: Se enriquecerán los esquemas Pydantic para generar documentación OpenAPI (Swagger) detallada, incluyendo descripciones de campos y ejemplos de respuesta, eliminando la ambigüedad de los contratos.
2. Mejora Estructural de IA	Robustecer el contrato de datos entre API e IA.	Incluye la implementación de tipado estricto y la normalización de salidas del modelo YOLO. Acción Documental: Se formalizará el esquema de los mensajes JSON que circulan por RabbitMQ, documentando explícitamente la estructura de los eventos de detección para facilitar futuras integraciones.
3. Seguridad Aplicada	Consolidar el modelo de seguridad Zero Trust.	Se enfocará en la implementación completa del flujo OAuth2, políticas de rotación de secretos y la integración de escaneo automático de vulnerabilidades en CI/CD.
4. Estrategia de Pruebas	Pagar la deuda de pruebas.	Consiste en la creación de una suite de pruebas unitarias para controladores, pruebas de integración para flujos críticos y establecimiento de umbrales de cobertura.
5. Desacoplamiento Físico y Gobierno	Transición a microservicios independientes y consolidación documental.	Ejecución de la fase final del patrón (Estrangulamiento Físico): Extracción de los módulos ya desacoplados en la Fase 1 hacia repositorios y contenedores autónomos. Al existir ya la separación lógica, esta fase se reduce a un cambio de infraestructura de bajo riesgo.

4.4 Gestión de Riesgos y Estrategia de Validación

Una restricción crítica identificada en el diagnóstico fue la ausencia de pruebas automatizadas (Cobertura 0.0%) y la falta de documentación explícita. Frente a esto, se adoptó un enfoque de **'Design for Testability'**, priorizando el saneamiento arquitectónico sobre la cobertura inmediata. Dado el alto acoplamiento del sistema legado, la implementación prematura de pruebas unitarias habría generado **'tests frágiles'**, costosos de mantener y de bajo valor. Para mitigar el riesgo de regresión durante esta transición, se formalizó un protocolo de **validación de flujos críticos** a nivel de sistema. Esta estrategia de caja negra permitió asegurar la integridad funcional antes de contar con la infraestructura necesaria para la automatización.

Para operacionalizar esta estrategia y mitigar los riesgos durante la refactorización (Fases 0-2), se formalizaron las siguientes acciones:

- **Protocolo de Validación en Staging:** Ante la falta de pruebas automáticas, se estableció un procedimiento de revisión manual en dos etapas sobre el entorno de pre-producción:
 - **Smoke Testing (Pruebas de Humo):** Verificación inicial de despliegue y conectividad entre contenedores (API, RabbitMQ, BD) para descartar fallos de configuración.
 - **Pruebas de Regresión Manuales:** Ejecución sistemática de los casos de uso principales (autenticación, ingesta de imágenes, inferencia) antes de cada integración (merge), garantizando que la refactorización no alterara la lógica de negocio existente
- **Verificación de Compatibilidad de Contratos:** Se validó la estricta equivalencia de las respuestas JSON entre la versión *legacy* y la refactorizada (Backwards Compatibility). Se utilizaron los esquemas OpenAPI generados para auditar que la estructura de datos, tipos de campo y códigos de estado HTTP se mantuvieran inalterados para los clientes móviles
- **Documentación como Código:** Para mitigar el riesgo de conocimiento tácito ("Factor de Bus") , se exigió que cada refactorización incluyera su correspondiente actualización en los esquemas de autodocumentación (OpenAPI), asegurando que el conocimiento técnico quedara explicitado en el repositorio.

5 Implementación

La implementación de la primera iteración del plan de mejora se ejecutó siguiendo una estrategia de refactorización incremental, priorizando la estabilidad del sistema y la mitigación de riesgos de seguridad. Este capítulo detalla el entorno técnico, las intervenciones específicas realizadas en cada fase y el protocolo de validación aplicado para asegurar la calidad sin contar con pruebas automatizadas previas, reorganización modular de la API y núcleo común de infraestructura.

5.1 Entorno de Desarrollo y Protocolo de Despliegue

Para garantizar la reproducibilidad y el aislamiento de las intervenciones, se estableció un **entorno de desarrollo contenerizado** basado en **Docker** y **Docker Compose** (ver configuración de infraestructura en el **Anexo B**). Esta decisión permitió replicar la arquitectura distribuida (API, IA, Base de Datos, Broker) en un entorno local controlado, mitigando las discrepancias de configuración entre entornos de desarrollo y producción y asegurando la portabilidad del sistema.

El flujo de trabajo se estructuró bajo prácticas de control de versiones:

1. **Aislamiento de Cambios:** Cada fase del plan de mejora se abordó en ramas de características (feature branches) específicas.
2. **Orquestación de Servicios:** Se configuró docker-compose para levantar la infraestructura completa (MongoDB, RabbitMQ, API, IA) con un solo comando, actuando como el **entorno de integración** donde se verificaron las interacciones entre servicios.
3. **Gestión de Configuración:** Se externalizaron todas las variables de entorno sensibles (credenciales, hosts, tokens) mediante archivos .env no versionados.

5.2 Ejecución de la Fase 0: Hardening y Seguridad Transversal

Antes de modificar la estructura del código, se aplicó una capa de "endurecimiento" (*hardening*) para cerrar las brechas de seguridad más críticas detectadas en el diagnóstico.

- **Saneamiento de Cabeceras HTTP:** Se configuró el middleware de **CORS** en FastAPI para restringir los orígenes permitidos (Allow-Origins) únicamente a los clientes web autorizados, eliminando la política permisiva (*). Adicionalmente, se forzaron cabeceras de seguridad básicas en las respuestas.



- **Gestión Segura de Secretos:** Se refactorizó el arranque de la aplicación para validar la existencia y complejidad de SECRET_KEY antes de iniciar el servidor. Si la clave es débil o inexistente, el servicio se detiene inmediatamente (Fail-Fast), previniendo el despliegue de instancias inseguras.
- **Protección de Credenciales:** Se sustituyó el almacenamiento de contraseñas en texto plano por un esquema de hashing robusto utilizando **Bcrypt**. Se migró la base de datos de usuarios existente y se definieron hooks para asegurar que ningún nuevo usuario pudiera registrarse sin encriptación.

5.3 Ejecución de la Fase 1: Refactorización Modular y Documentación Viva

Esta fase abordó la **deuda arquitectónica** y la **deuda de documentación** simultáneamente, aplicando principios de diseño modular.

5.3.1 Reorganización por Dominios (Descomposición Lógica)

Se transformó la estructura de directorios monolítica hacia una organización por dominios funcionales (users, roads, files).

- **Separación de Capas:** Dentro de cada dominio, se segregaron las responsabilidades en:
 - schemas.py: Definición de contratos de datos.
 - router.py: Manejo de peticiones HTTP.
 - controllers.py: Lógica de negocio pura.
 - queries.py: Interacción con la base de datos.
- **Desacoplamiento:** Se eliminaron las importaciones circulares y las dependencias directas entre rutas y base de datos, facilitando la lectura y futura *testabilidad* del código.

5.3.2 Generación de Documentación Viva (Living Documentation)

Para pagar la deuda de documentación, se aprovechó la capacidad de introspección de **FastAPI** junto con **Pydantic**.

- **Tipado Estricto:** Se redefinieron todos los modelos de entrada y salida (Pydantic Models) con tipos de datos estrictos (StrictStr, PositiveInt) y validaciones de campo (Field).
- **Enriquecimiento de Metadatos:** Se añadieron descripciones (description), ejemplos (example) y títulos a cada campo de los modelos.
- **Resultado Automático:** Como consecuencia directa de esta refactorización de código, la documentación **OpenAPI (Swagger UI)** se regeneró automáticamente (ver comparativa de evolución de contratos en el **Anexo C**), pasando de ser un listado de endpoints vacíos a una referencia interactiva completa con esquemas detallados, códigos de error y ejemplos de uso real. Esto garantiza que la documentación nunca se desincronice del código.

5.4 Ejecución de la Fase 2: Robustez en el Servicio de IA

La intervención en el microservicio de IA se centró en la **Fiabilidad** y la estabilidad del contrato de datos.

- **Manejo de Mensajería (RabbitMQ):** Se implementó un bloque try/except robusto en el consumidor de mensajes. Se configuró el envío de la confirmación (ch.basic_ack) solo tras el procesamiento exitoso de la imagen. En caso de error, el mensaje se *re-encola* o se descarta según la política de reintentos, evitando la pérdida silenciosa de datos o el bloqueo del worker.
- **Normalización de Salidas:** Se estandarizó el formato de respuesta del modelo YOLO. Anteriormente, el sistema devolvía índices numéricos propensos a error; se refactorizó para devolver etiquetas de cadena normalizadas (ej: "bache", "grieta") validadas contra un *Enum* compartido con la API, eliminando inconsistencias de datos.

5.5 Estrategia de Validación y Mitigación de Regresiones

Dado que el proyecto no contaba con una suite de pruebas automatizada (Cobertura 0%), se ejecutó un protocolo de validación manual riguroso sobre el **entorno de integración (Docker Compose)** para mitigar el riesgo de regresión.

1. **Validación de Flujos Críticos:** Antes de integrar cualquier cambio en la rama principal, se levantó el entorno completo en local y se ejecutó una batería de **pruebas de regresión manuales**:
 - *Flujo de Autenticación:* Login exitoso, rechazo de credenciales inválidas y expiración de tokens.
 - *Ciclo de Vida de Imagen:* Subida desde la App Móvil → Procesamiento en Cola → Inferencia IA → Persistencia en BD → Visualización en Mapa.
2. **Verificación de Contratos de API:** Se utilizó la documentación interactiva (Swagger) generada en la Fase 1 para enviar payloads de prueba y verificar que las respuestas JSON mantuvieran la estructura esperada por el cliente móvil, asegurando compatibilidad hacia atrás (Backwards Compatibility).
3. **Validación de Resiliencia:** Se simuló caídas del servicio de IA (deteniendo el contenedor Docker) para verificar que la API no colapsara y que los mensajes quedaran encolados en RabbitMQ hasta la recuperación del servicio, validando las mejoras de fiabilidad implementadas.

Este enfoque de validación sobre un entorno efímero idéntico a producción permitió detectar y corregir errores de integración de manera temprana, compensando la falta de automatización en esta etapa inicial del plan de mejora.

6 Resultados

La evaluación de la primera iteración del plan de mejora se realizó replicando rigurosamente la metodología de diagnóstico inicial. Se compararon las métricas de línea base ("Antes") establecidas en el diagnóstico contra el estado final del sistema ("Después"), utilizando los reportes generados por las herramientas SQA seleccionadas (ver evidencia gráfica comparativa en el **Anexo D**) tras la ejecución de las fases de refactorización y *hardening*.

Los resultados se presentan estructurados según las dimensiones de calidad del modelo ISO/IEC 25010:2023: Mantenibilidad, Seguridad y Adecuación Funcional.

6.1 Resultados de Calidad Interna (Mantenibilidad)

Esta dimensión evalúa cómo las intervenciones estructurales han reducido la fricción para futuros desarrollos y eliminado la deuda técnica acumulada.

6.1.1 SonarQube: Complejidad y Deuda Técnica

Se ejecutó un nuevo análisis estático sobre la base de código refactorizada. La siguiente tabla resume el impacto de la reorganización modular y la limpieza de código.

Tabla 10: Comparación de resultados SonarQube antes y después

Fuente: Elaboración propia

Métrica	Antes	Después	Impacto en Calidad
Reliability Rating	E (4 bugs)	A (0 bugs)	Eliminación de defectos estáticos críticos; reducción preventiva de vectores de fallo y mejora en la robustez del código.
Maintainability	A (49 smells)	A (0 smells)	Código limpio; reducción del ruido cognitivo para desarrolladores.
Deuda Técnica	~5 horas	0 min	Amortización de los code smells y defectos críticos identificados por análisis estático en la línea base. Representa



			un estado de higiene de código óptima para iniciar la fase de pruebas automatizadas sin arrastrar vicios lógicos previos.
Complejidad Cognitiva	173	125 (-28%)	El flujo de control es más lineal y fácil de entender.
Complejidad Ciclomática	180	192 (+6%)	Incremento nominal asociado a la creación de nuevas capas de abstracción y lógica de desacoplamiento. Este aumento sistémico es una consecuencia esperada de la transición hacia una arquitectura modular, priorizando la segregación de responsabilidades y la claridad estructural.

La eliminación de la deuda técnica de código detectable mediante análisis estático valida la **fiabilidad estática** del sistema. Esto confirma un estado de 'higiene de código' óptimo, eliminando defectos lógicos y condiciones de carrera potenciales. Sin embargo, se reconoce que esta métrica no garantiza por sí sola la estabilidad en tiempo de ejecución, la cual se aborda mediante la validación de la arquitectura asíncrona. Asimismo, la caída del 28% en la complejidad cognitiva es el indicador principal de una mayor *analizabilidad*, facilitando la comprensión de los flujos lógicos. Si bien la complejidad ciclomática total subió un 6% debido a la mayor cantidad de componentes del sistema, la redistribución de la lógica en unidades pequeñas compensa este aumento. Este fenómeno transforma el código en una estructura de alta granularidad, permitiendo un mantenimiento mucho más focalizado y eficiente.

6.1.2 Pylint: Estandarización

Tabla 11: Comparación de puntuaciones Pylint

Fuente: Elaboración propia

Proyecto	Antes	Después	Variación
IA	5.56	8.34	50%
API	5.56	7.95	43%

El incremento sustancial en la puntuación refleja la adopción de estándares PEP-8 y la incorporación sistemática de docstrings. El código ahora es auto-documentado a nivel de implementación, cumpliendo con el criterio de *analizabilidad* de la norma ISO/IEC 25010:2023.

6.1.3 Emerge-viz: Desacoplamiento Estructural

Las métricas estructurales permiten visualizar el efecto de la estrategia Strangler Fig en su primera etapa tanto en la API como en el servicio de IA.

Tabla 12: Métricas Emerge-viz antes y después de la refactorización

Fuente: Elaboración propia

Métrica	API Antes	API Después	IA Antes	IA Después	Interpretación
SLOC Promedio	146	20.9	83	35.5	Reducción masiva (-85% en API). Archivos pequeños y cohesivos (Principio de Responsabilidad Única).
Fan-out Máximo	20	14	8	9	Reducción crítica del acoplamiento en la API. El leve aumento en IA responde a la normalización de utilidades compartidas.
Métodos por archivo	9.25	1.76	3.71	1.17	Granularidad fina en ambos servicios, facilitando las pruebas unitarias futuras.

Los resultados de la Tabla 12 justifican el leve incremento de complejidad ciclomática reportado anteriormente. La reducción masiva del SLOC promedio en la API (-85%, pasando de 146 a 21 líneas) demuestra la transición exitosa desde un 'monolito por directorio' hacia una estructura modular limpia donde la complejidad no creció

de forma negativa, sino que se distribuyó en componentes atómicos. Este es el 'costo' necesario de una arquitectura orientada a dominios: se obtiene una estructura total más robusta, pero con piezas individuales significativamente más sencillas de mantener, garantizando que la *modificabilidad* sea ahora una propiedad intrínseca del backend.

6.2 Resultados de Seguridad (Hardening)

6.2.1 Análisis Estático y Dependencias

La intervención de seguridad se valida mediante la comparación directa de hallazgos antes y después del *hardening*.

Tabla 13: Comparativa de Hallazgos Bandit (SAST)

Fuente: Elaboración propia

Proyecto	Antes	Después	Estado
IA	1 (Media)	0	Corregido. Se añadieron timeouts explícitos a las llamadas externas.
API	1 (Baja)	0	Corregido. Eliminación de credenciales <i>hardcodeadas</i> .

Tabla 14: Comparativa de Vulnerabilidades (Dependency-Check)

Fuente: Elaboración propia

Proyecto	Antes	Después	Estado
IA	1 (Crítica)	0	Mitigado. Actualización de PyTorch a versión segura.
API	0	0	Mantenimiento de estado seguro.

La limpieza total de alertas en Bandit y Dependency-Check certifica que el código base ya no contiene patrones inseguros conocidos ni dependencias obsoletas, cumpliendo el objetivo de la Fase 0.

6.2.2 OWASP ZAP: Seguridad Perimetral

El análisis dinámico validó la eficacia de la configuración de cabeceras en el servidor de aplicaciones.

Tabla 15: Evolución de Alertas OWASP ZAP

Fuente: Elaboración propia

Severidad	Antes	Después	Estado
Media	6	0	Mitigado. CORS restrictivo y cabeceras de seguridad implementadas.
Baja	4	1	Riesgo residual aceptable (configuraciones informativas).

La eliminación de todas las alertas de severidad media garantiza que la API ahora rechaza orígenes no autorizados y protege a los clientes contra ataques básicos de XSS y *Clickjacking*.

6.3 Resultados de Calidad Externa y Funcional

Más allá de las métricas de código, se validó que el sistema siguiera operando correctamente y que la documentación fuera útil para el consumo.

6.3.1 Validación Funcional (Pruebas de Sistema y Regresión)

Para asegurar que la refactorización no introdujera regresiones, se ejecutó un protocolo de pruebas manuales sistemático sobre el entorno de integración, cubriendo los flujos críticos de negocio (ver protocolo y evidencia de ejecución en el **Anexo E**). Este protocolo contempló una batería de aproximadamente 2 a 4 casos de prueba



por endpoint crítico, cubriendo tanto el flujo exitoso (*happy path*) como casos de borde (datos inválidos, errores de red). Adicionalmente, se modeló un **escenario de fallo controlado** para validar las subcaracterísticas de tolerancia a fallos y recuperabilidad. Este escenario consistió en la interrupción forzada del contenedor de IA durante una carga de trabajo activa, verificando que:

1. La API mantiene su disponibilidad (respondiendo 200 OK).
2. Los mensajes persisten seguros en la cola de RabbitMQ (Integridad).
3. El procesamiento se reanuda automáticamente al restablecer el servicio (Auto-recuperación).

Tabla 16: Resultados del Protocolo de Validación

Fuente: Elaboración propia

Flujo Crítico	Cobertura	Resultado Esperado	Estado
Autenticación	3 casos (Login, Token inválido, Expirado)	Token JWT generado y validado correctamente según credenciales.	OK
Ingesta de Imagen	2 casos (Imagen válida, Archivo corrupto)	API acepta/rechaza y encola en RabbitMQ correctamente.	OK
Inferencia IA	4 casos (Detección positiva, Negativa, Timeout, Error formato)	Detección persistida en MongoDB con etiquetas normalizadas.	OK
Resiliencia	2 casos (IA caída, Reinicio de servicio)	API responde 202 Accepted; mensaje queda encolado y se procesa al recuperar IA.	OK

El sistema mantiene la integridad funcional en todos los flujos críticos evaluados, asegurando la compatibilidad hacia atrás en los casos de uso principales.

6.3.2 Validación de Documentación Viva

Como resultado de la adopción de esquemas Pydantic estrictos, la documentación OpenAPI (Swagger) experimentó una mejora cualitativa crítica:

- **Antes:** *Endpoints* opacos, sin esquemas de entrada/salida ("Any").
- **Después:** Contratos explícitos con tipos de datos, validaciones (ej: latitud entre -90 y 90) y ejemplos de respuesta. Esto reduce significativamente la **deuda de documentación**, transformando la API de una "caja negra" a un servicio explícito y consumible.

6.4 Análisis General de Impacto

La intervención ejecutada ha transformado el perfil de calidad del backend de Geoviality.

1. **Fiabilidad:** Se complementó la mejora del rating estático ('E' a 'A') con la validación de la tolerancia a fallos operativa, demostrando que la arquitectura desacoplada absorbe caídas críticas del servicio de IA sin interrumpir la disponibilidad del sistema.
2. **Seguridad:** Se cerraron vectores de ataque web (OWASP) y se sanearon las dependencias, estableciendo una línea base segura.
3. **Mantenibilidad:** La amortización de la deuda técnica (casi 5 horas estimadas) y la reducción de complejidad cognitiva dejan el código preparado para evolucionar.

Si bien la **deuda de pruebas automáticas** persiste (cobertura 0%), la estructura actual del código (métodos pequeños y desacoplados) hace que el sistema sea ahora **altamente testeable**, dejando el terreno preparado para la implementación eficiente de la Fase 4 del plan de mejora.

7 Conclusiones

El desarrollo de esta memoria ha permitido abordar una problemática común en la industria del software: la evolución de un prototipo académico funcional ("Time-to-Market") hacia un producto de ingeniería sostenible. A través de la sistematización del pago de la deuda técnica en el backend de Geoviality, se validó la hipótesis

de que es posible recuperar la mantenibilidad y seguridad de un sistema legado sin recurrir a una reescritura total, aplicando estrategias de refactorización incremental y validación controlada.

7.1 Cumplimiento de Objetivos y Resultados

Se lograron avances medibles y cuantificables en la calidad del software, validando el cumplimiento del objetivo general mediante la transformación de un "monolito distribuido" frágil en una arquitectura modular orientada a dominios.

- **Reducción de Deuda:** Se constató el **saneamiento de la deuda técnica estática** y de seguridad identificada en el diagnóstico (reducción de ~5 horas a 0 minutos en SonarQube). Si bien la cobertura de pruebas automatizadas se definió fuera del alcance de esta primera iteración, la refactorización logró transformar la deuda de *testabilidad* (código imposible de probar) en una tarea de implementación ejecutable para la siguiente fase, al eliminar la complejidad ciclomática que impedía la implementación de tests efectivos.
- **Mejora de Calidad (ISO/IEC 25010:2023):** Las métricas obtenidas evidencian una elevación en los estándares de **mantenibilidad** (reducción de la complejidad cognitiva en un 28%) y **seguridad** (mitigación de alertas OWASP), estableciendo una base técnica sólida para el crecimiento futuro del sistema.

7.2 Principales Contribuciones e Impacto

Más allá de la limpieza del código, esta memoria aporta contribuciones metodológicas y técnicas relevantes para el contexto de desarrollo de software en entornos con recursos limitados:

1. **Metodología de Refactorización en Escenarios de Baja Cobertura:** Se demostró la viabilidad técnica de refactorizar sistemas con 0% de cobertura de pruebas mediante la formalización de un protocolo de **validación de flujos críticos en entorno contenerizado**. Esta estrategia sirve de referencia para proyectos que deban pagar deuda arquitectónica crítica antes de contar con la madurez para implementar automatización completa.
2. **Documentación Viva como Activo de Ingeniería:** La implementación de esquemas estrictos (Pydantic) para generar documentación automática (OpenAPI) probó ser un mecanismo eficiente para mitigar el riesgo de pérdida de conocimiento ("Factor de Bus"), transformando una "caja negra" en una interfaz explícita y consumible sin incurrir en sobrecarga administrativa.
3. **Elevación del Nivel de Madurez del Producto:** Esta intervención permitió que Geoviality transitara desde un prototipo académico (enfocado solo en demostrar que la detección funciona) hacia un **Producto Mínimo Viable (MVP) robusto**. Al incorporar manejo de errores resiliente, seguridad perimetral y estandarización de contratos, el sistema ahora cumple con los requisitos no funcionales necesarios para operar en un entorno municipal real, reduciendo la brecha entre la innovación universitaria y la transferencia tecnológica comercial.

7.3 Alcances y Limitaciones del Trabajo

Es fundamental reconocer las fronteras de esta intervención para contextualizar adecuadamente los resultados obtenidos:

- **Sostenibilidad de la Validación Manual:** Si bien el protocolo de validación manual sistemática resultó efectivo para esta primera iteración, se identifica como una limitación para la escalabilidad del proyecto. A medida que la complejidad del sistema aumente, la dependencia de pruebas manuales se convertirá en un cuello de botella, por lo que la implementación de una suite de pruebas automatizada (CI/CD) se identificó como el siguiente paso lógico después de la primera iteración. La validación manual, aunque efectiva para esta etapa de reestructuración, no es escalable a largo plazo, por lo que la automatización constituye la prioridad inmediata para el trabajo futuro.



- **Alcance de la Seguridad:** El *hardening* implementado cerró brechas evidentes de configuración y dependencias, estableciendo una línea base de seguridad. Sin embargo, esto no sustituye la necesidad de auditorías de seguridad profundas o pruebas de penetración (*Pentesting*) profesionales para un entorno productivo de alta exposición.
- **Condiciones del Entorno de Validación:** Los resultados se verificaron en un entorno de integración local contenerizado. El comportamiento del sistema bajo condiciones de estrés, latencia de red variable y concurrencia masiva no fue evaluado en esta etapa, dejando el atributo de **eficiencia de desempeño** como una incógnita a despejar antes de un despliegue masivo.

7.4 Recomendaciones y Trabajo Futuro

Para dar continuidad a la línea de investigación y desarrollo, se proponen las siguientes acciones priorizadas:

1. **Automatización de Pruebas (Prioridad Crítica):** Antes de la incorporación de nuevas funcionalidades, resulta imperativo implementar la **Fase 4** del plan de mejora. Se recomienda iniciar con pruebas de integración (API → RabbitMQ → IA), dado que aportan el mayor valor por esfuerzo invertido en arquitecturas distribuidas.
2. **Observabilidad:** Se sugiere la implementación de un stack de monitoreo para visualizar métricas de negocio y técnicas en tiempo real, permitiendo transitar de un enfoque reactivo a uno proactivo en la gestión de incidentes.
3. **Complejidad del Patrón Strangler (Extracción Física):** Con la deuda técnica saneada y los módulos definidos, se recomienda la extracción del servicio de 'Gestión de Vialidad' a su propio contenedor. Esto eliminará la última dependencia de estado local, permitiendo que el monolito modular se divida finalmente en microservicios escalables, completando la arquitectura TO-BE.

7.5 Reflexiones Finales sobre la Ingeniería de Software

Desde una perspectiva formativa, la ejecución de esta memoria marca una transición en la aproximación a la disciplina: el paso de la codificación funcional hacia el diseño de sistemas sostenibles.

Se evidenció que la **deuda técnica** trasciende la metáfora teórica, constituyendo un obstáculo tangible que paraliza la innovación y aumenta el riesgo operativo. El enfrentamiento con una base de código carente de documentación y pruebas resaltó el valor de la empatía técnica en la ingeniería de software: el código debe escribirse no solo para ser ejecutado por máquinas, sino para ser mantenido por personas. La decisión estratégica de priorizar la calidad interna y la estructura sobre la adición de nuevas funcionalidades reafirma que la excelencia técnica no es un lujo, sino el único camino viable para mantener la velocidad de entrega y la fiabilidad del software a largo plazo.

Agradecimientos. Quiero agradecer primero a mi familia, mis padres, por su apoyo desde que fui niño, hasta ya de adulto en que me convierta en un profesional, a mis abuelos por estar ahí con amor incondicional, a mis niñas Luna y Perla, por mostrarme las cosas que realmente importan, y también a mis amigos, Max y Cris, por siempre estar ahí, en cada trabajo, colaborando y dándolo todo como un equipo, y por último a Camila, mi novia, por siempre creer en mí y acompañarme en todo este proceso desde que estamos juntos.

Referencias

- [1] "Geoviality - Innovación en Gestión Vial." Accessed: Dec. 07, 2025. [Online]. Available: <https://geoviality.feriadesoftware.cl/>
- [2] "FastAPI." Accessed: Dec. 10, 2025. [Online]. Available: <https://fastapi.tiangolo.com/>
- [3] G. Jocher, *YOLOv5 by Ultralytics*. (May 2020). Python. doi: 10.5281/zenodo.3908559.
- [4] "RabbitMQ: One broker to queue them all | RabbitMQ." Accessed: Dec. 10, 2025. [Online]. Available: <https://www.rabbitmq.com/>



- [5] ISO/IEC 25010: 2023, “Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuARE)—Product Quality Model.” International Organization for Standardization Geneva, Switzerland, 2023.
- [6] A. Pellon, “Veredas y calles en mal estado: casos en los que me corresponde una indemnización por daños | BBCL Contigo | BioBioChile.” Accessed: Dec. 07, 2025. [Online]. Available: <https://www.biobiochile.cl/noticias/servicios/explicado/2023/12/15/calle-vereda-en-mal-estado-como-solicitar-indemnizacion.shtml>
- [7] “Diagnóstico de Siniestros de Tránsito en Chile 2024,” Comisión Nacional de Seguridad de Tránsito (CONASET), Chile, Sept. 2024. Accessed: Dec. 07, 2025. [Online]. Available: <https://www.conaset.cl/wp-content/uploads/2025/09/Chile-2024.pdf>
- [8] “Homepage | SonarQube Server | Sonar Documentation.” Accessed: Dec. 07, 2025. [Online]. Available: <https://docs.sonarsource.com/sonarqube-server>
- [9] “Pylint - code analysis for Python | www.pylint.org.” Accessed: Dec. 07, 2025. [Online]. Available: <https://www.pylint.org/>
- [10] *PyCQA/bandit*. (Dec. 07, 2025). Python. Python Code Quality Authority. Accessed: Dec. 07, 2025. [Online]. Available: <https://github.com/PyCQA/bandit>
- [11] “OWASP Dependency-Check | OWASP Foundation.” Accessed: Dec. 07, 2025. [Online]. Available: <https://owasp.org/www-project-dependency-check/>
- [12] “The ZAP Homepage,” ZAP. Accessed: Dec. 07, 2025. [Online]. Available: <https://www.zaproxy.org/>
- [13] S. H. Abdelwahed, B. K. Sharobim, B. Wasfey, and L. A. Said, “Advancements in real-time road damage detection: a comprehensive survey of methodologies and datasets,” *J. Real-Time Image Process.*, vol. 22, no. 4, pp. 1–13, 2025.
- [14] H. Maeda, Y. Sekimoto, T. Seto, T. Kashiya, and H. Omata, “Road damage detection and classification using deep neural networks with smartphone images,” *Comput.-Aided Civ. Infrastruct. Eng.*, vol. 33, no. 12, pp. 1127–1141, 2018.
- [15] A. Mednis, G. Strazdins, R. Zviedris, G. Kanonirs, and L. Selavo, “Real time pothole detection using android smartphones with accelerometers,” in *2011 International conference on distributed computing in sensor systems and workshops (DCOSS)*, IEEE, 2011, pp. 1–6.
- [16] S. Sattar, S. Li, and M. Chapman, “Road surface monitoring using smartphone sensors: A review,” *Sensors*, vol. 18, no. 11, p. 3845, 2018.
- [17] A. Faisal and S. Gargoum, “Cost-effective LiDAR for pothole detection and quantification using a low-point-density approach,” *Autom. Constr.*, vol. 172, p. 106006, 2025.
- [18] “PyTorch documentation — PyTorch 2.9 documentation.” Accessed: Dec. 13, 2025. [Online]. Available: <https://docs.pytorch.org/docs/stable/index.html>
- [19] “MongoDB Documentation - Homepage.” Accessed: Dec. 12, 2025. [Online]. Available: <https://www.mongodb.com/docs/>
- [20] “Docker: Accelerated Container Application Development.” Accessed: Dec. 10, 2025. [Online]. Available: <https://www.docker.com/>
- [21] W. Cunningham, “The WyCash portfolio management system,” *ACM Sigplan Oops Messenger*, vol. 4, no. 2, pp. 29–30, 1992.
- [22] Roger S Pressman, *A practitioner’s approach*, vol. 2. 2010.
- [23] Richards, Mark and Ford, Neal, *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA: O’Reilly Media, 2020.
- [24] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA: O’Reilly Media, 2021.
- [25] “OWASP Top 10:2025.” Accessed: Dec. 13, 2025. [Online]. Available: <https://owasp.org/Top10/2025/>
- [26] “PEP 8 – Style Guide for Python Code | peps.python.org,” Python Enhancement Proposals (PEPs). Accessed: Dec. 29, 2025. [Online]. Available: <https://peps.python.org/pep-0008/>
- [27] G. Lato, *glato/merge*. (Dec. 07, 2025). Python. Accessed: Dec. 07, 2025. [Online]. Available: <https://github.com/glato/merge>
- [28] “Postman documentation overview,” Postman Docs. Accessed: Dec. 29, 2025. [Online]. Available: <https://learning.postman.com/docs/introduction/overview/>



Anexos

Anexo A: Bitácora Detallada de Revisión Manual de Código y Arquitectura

El presente anexo documenta la totalidad de los hallazgos levantados durante la inspección manual del código fuente. Esta revisión complementa el análisis estático automatizado, identificando deuda arquitectónica, violaciones de principios de diseño y riesgos semánticos específicos.

Se identificaron un total de **26 patrones de defectos** distribuidos en los distintos módulos del sistema, clasificados según su impacto en el modelo de calidad ISO/IEC 25010:2023.

Componente: API Backend

Tabla A 1: Registro de hallazgos en el servicio de API

Fuente: Elaboración propia

ID	Hallazgo (Descripción del Defecto)	Ubicación / Instancias Detectadas	Severidad	Tipo de Deuda	Principio ISO/IEC 25010:2023 Afectado
API-01	Rutas de sistema de archivos Hardcodeadas. Uso de rutas absolutas dependientes del SO local (ej: C:/Users/...) que rompen la portabilidad.	3 Archivos: routes.py, utils.py, config.py	Alta	Arquitectura	Portabilidad (Adaptabilidad)
API-02	Lógica invertida en CRUD. Manejo incorrecto de booleanos donde se interpreta éxito como error y viceversa en respuestas HTTP.	1 Archivo: controllers.py (Módulo crítico de usuarios)	Alta	Código	Adecuación Funcional (Corrección)
API-03	Exposición de Secretos (Hardcoded Secrets). La variable SECRET_KEY para firma de tokens JWT está escrita explícitamente en el código.	1 Archivo: auth.py	Crítica	Seguridad	Seguridad (Confidencialidad)
API-04	Almacenamiento de contraseñas en texto plano. Persistencia de credenciales sin aplicar funciones de Hashing (bcrypt/argon2).	1 Archivo: models.py (Modelo User)	Crítica	Seguridad	Seguridad (Protección de datos)
API-05	Acoplamiento de Lógica en Enrutador. Funciones de ruta (@app.post) contienen lógica de negocio, validación y acceso a datos mezclados.	2 Archivos: routes.py, main.py	Media	Arquitectura	Mantenibilidad (<i>Modularidad</i>)
API-06	Captura genérica de excepciones. Uso de try/except Exception vacíos o genéricos que ocultan la causa raíz de los errores.	4 Archivos: routes.py, crud.py, auth.py, utils.py	Alta	Fiabilidad	Fiabilidad (Tolerancia a fallos)
API-07	Estado mutable en variables globales. Uso de listas globales para almacenar estados temporales entre peticiones.	1 Archivo: routes.py (Variable processed_images)	Alta	Arquitectura	Fiabilidad (Tolerancia a fallos)
API-08	Validación laxa de entradas (Input Validation). Modelos Pydantic usando tipos Any o sin restricciones de longitud/formato.	2 Archivos: schemas.py (Modelos de entrada), routes.py	Alta	Seguridad	Seguridad (Integridad)

API-09	E/S Bloqueante en entorno Asíncrono. Uso de drivers síncronos (pymongo) dentro de funciones async def.	1 Archivo: database.py	Media	Arquitectura	Eficiencia de Desempeño
API-10	Ausencia de Transacciones Atómicas. Operaciones de escritura múltiples sin mecanismos de rollback ante fallos parciales.	1 Archivo: crud.py	Alta	Fiabilidad	Fiabilidad (<i>Recuperabilidad</i>)
API-11	Dependencia de event_queue inexistente. El endpoint de WebSocket referencia una cola no inicializada correctamente.	1 Archivo: routes.py	Alta	Arquitectura	Fiabilidad (<i>Disponibilidad</i>)
API-12	Endpoint de datos expuesto. La ruta /data/processed_image permite acceso sin validación estricta de token.	1 Archivo: routes.py	Alta	Seguridad	Seguridad (<i>Autenticidad</i>)
API-13	Números Mágicos. Uso de literales numéricos en lógica condicional sin contexto semántico.	2 Archivos: utils.py, routes.py	Baja	Código	Mantenibilidad (<i>Analizabilidad</i>)
API-14	Ausencia Sistemática de Docstrings. Falta de documentación interna en clases y métodos complejos.	Transversal: Detectado en >80% de los archivos del backend.	Media	Documentación	Mantenibilidad (<i>Analizabilidad</i>)
API-15	Exceso de importaciones no utilizadas. Cabeceras con librerías importadas que no se usan ("Ruido visual").	3 Archivos: routes.py, models.py, schemas.py	Baja	Código	Mantenibilidad (<i>Modificabilidad</i>)

Componente: Servicio de IA

Tabla A 2: Registro de hallazgos en el servicio de procesamiento de imágenes

Fuente: Elaboración propia

ID	Hallazgo (Descripción del Defecto)	Ubicación / Instancias Detectadas	Severidad	Tipo de Deuda	Principio ISO/IEC 25010:2023 Afectado
IA-01	Confirmación prematura (Auto-Ack). El consumidor RabbitMQ envía ack antes de procesar, provocando pérdida de mensajes si hay error.	1 Archivo: main.py	Crítica	Arquitectura	Fiabilidad (<i>Recuperabilidad</i>)
IA-02	Uso de exit() ante errores. El script termina el proceso del contenedor abruptamente si un archivo es inválido, en lugar de manejar la excepción.	1 Archivo: ia_predictor.py	Crítica	Arquitectura	Fiabilidad (<i>Disponibilidad</i>)
IA-03	Conversión insegura a GeoJSON. Transformación directa de datos de BD a JSON sin validar valores None (Null Safety).	1 Archivo: funcs.py	Alta	Código	Fiabilidad (<i>Robustez</i>)
IA-04	Cálculo incorrecto de confianza. Multiplicación por 100 de la confianza y comparación contra umbral 0-1 (Lógica errónea).	1 Archivo: ia_predictor.py	Media	Código	Adecuación Funcional (<i>Exactitud</i>)
IA-05	Dependencia de rutas locales. Carga del modelo .pt referenciando rutas absolutas de la máquina de desarrollo.	1 Archivo: detector_ia.py	Alta	Arquitectura	Portabilidad (<i>Instalabilidad</i>)



IA-06	Inicialización costosa en bucle. Carga del modelo YOLO en memoria por cada mensaje procesado (No Singleton).	1 Archivo: main.py	Alta	Arquitectura	Eficiencia de Desempeño
IA-07	Llamada HTTP sin Timeout. Peticiones a servicios externos sin tiempo de espera definido (Riesgo de bloqueo indefinido).	1 Archivo: funcs.py	Media	Seguridad	Fiabilidad (Disponibilidad)
IA-08	Conexiones a BD ineficientes. Apertura y cierre de conexión a Mongo por cada mensaje individual.	1 Archivo: funcs.py	Media	Arquitectura	Eficiencia de Desempeño
IA-09	Logging mediante print. Depuración por salida estándar, imposibilitando el filtrado de logs en producción.	2 Archivos: detector_ia.py, main.py	Media	Código	Mantenibilidad (<i>Analizabilidad</i>)
IA-10	Descarga de pesos en Runtime. Intento de descarga de archivos del modelo desde internet en cada arranque.	1 Archivo: ia_predictor.py	Media	Código	Eficiencia de Desempeño
IA-11	Dependencias Vulnerables. Uso de versiones de librerías (torch, opencv) con CVEs reportados.	1 Archivo: requirements.txt	Alta	Seguridad	Seguridad (Integridad)

Anexo B: Configuración del Entorno de Despliegue Containerizado

Este anexo documenta la infraestructura como código (IaC) implementada para garantizar la **portabilidad**, **eficiencia** y **seguridad** del sistema. Se detalla la orquestación de servicios y la definición de imágenes optimizadas para producción.

Orquestación de Servicios (Docker Compose)

```
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - "5601:5672"
      - "15601:15672"
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
      TZ: America/Santiago
  api:
    build:
      context: ./geoviality-api
    container_name: api-container
    ports:
      - "8080:8080"
volumes:
  - ./geoviality-api:/app
  - api_services:/app/services
environment:
  TZ: America/Santiago
env_file:
  - ./geoviality-api/.env
depends_on:
  - rabbitmq
  - mongo
restart: unless-stopped
healthcheck:
```

```
test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
interval: 30s
timeout: 5s
retries: 5
start_period: 10s
ia:
  build:
    context: ./geoviality-ia
  container_name: ia-container
  volumes:
    - ./geoviality-ia/ia_service:/app/ia_service
    - ia_imgs:/app/ia_service/imgs
  environment:
    TZ: America/Santiago
  env_file:
    - ./geoviality-ia/.env
  depends_on:
    - rabbitmq
    - mongo

mongo:
  image: mongo:latest
  container_name: mongo
  ports:
    - "27018:27017"
  environment:
    MONGO_INITDB_ROOT_USERNAME: ${MONGO_USER}
    MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASSWORD}
    TZ: America/Santiago
  volumes:
    - mongo-data:/data/db

volumes:
  mongo-data:
  ia_imgs:
  api_services:
```

Definición de Imágenes Optimizadas (Dockerfiles)

Las imágenes de contenedor se diseñaron siguiendo principios de seguridad (*hardening*) y eficiencia de almacenamiento.

Dockerfile API: Se implementa la ejecución mediante un usuario de sistema sin privilegios (app) para mitigar riesgos de seguridad, evitando la ejecución como root (Principio de Menor Privilegio). Además, se configuran variables de entorno para optimizar el comportamiento de Python en contenedores.

```
FROM python:3.10-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

WORKDIR /app

RUN groupadd --system app && \
    useradd --system --gid app --home /app --shell /usr/sbin/nologin app

COPY requirements.txt /app/requirements.txt
```



```
RUN pip install --no-cache-dir --upgrade pip && \  
    pip install --no-cache-dir -r /app/requirements.txt  
  
COPY --chown=app:app . /app  
  
RUN mkdir -p /app/services/imgs && \  
    chown -R app:app /app/services  
  
EXPOSE 8080  
  
USER app  
  
CMD ["sh", "-c", "uvicorn app.main:app --host 0.0.0.0 --port ${PORT:-8080}"]
```

Dockerfile IA: Para este servicio, se utilizó una estrategia de construcción en etapas (**Multi-stage Build**). Esto permite compilar las dependencias en una etapa intermedia (builder) y copiar solo los binarios necesarios a la imagen final, reduciendo significativamente el peso del artefacto y eliminando herramientas de compilación innecesarias en producción.

```
FROM python:3.10-slim AS builder  
WORKDIR /build  
  
RUN apt-get update && apt-get install -y --no-install-recommends \  
    build-essential \  
    && rm -rf /var/lib/apt/lists/*  
  
COPY requirements.txt /build/requirements.txt  
RUN python -m pip install --no-cache-dir --upgrade pip wheel \  
    && pip wheel --no-cache-dir --wheel-dir /build/wheels -r  
/build/requirements.txt  
  
FROM python:3.10-slim  
WORKDIR /app  
  
RUN apt-get update && apt-get install -y --no-install-recommends \  
    libgl1 \  
    libglib2.0-0 \  
    libgomp1 \  
    libsm6 \  
    libxext6 \  
    libxrender1 \  
    && rm -rf /var/lib/apt/lists/*  
  
COPY --from=builder /build/wheels /wheels  
RUN python -m pip install --no-cache-dir --upgrade pip \  
    && pip install --no-cache-dir /wheels/* \  
    && rm -rf /wheels  
  
COPY ia_service/ /app/ia_service/  
  
RUN addgroup --system appgroup \  
    && adduser --system --ingroup appgroup --home /home/appuser appuser \  
    && chown -R appuser:appgroup /app /home/appuser  
  
ENV PYTHONUNBUFFERED=1 \  
    PYTHONDONTWRITEBYTECODE=1 \  
    HOME=/home/appuser
```



```
USER appuser
```

```
CMD ["python", "-m", "ia_service.app"]
```

Anexo C: Evolución de Contratos de API

Este anexo evidencia el pago de la **deuda de documentación** y la mejora en la **analizabilidad** del código (ISO/IEC 25010:2023). Se presenta una comparación directa entre la definición de esquemas de datos original (débilmente *tipada*) y la versión refactorizada (fuertemente *tipada* y auto-documentada).

Comparativa de Implementación de Esquemas

Versión Original (Antes): El código carecía de validaciones semánticas y metadatos. El campo email era un simple str, permitiendo cadenas inválidas, y no existía documentación integrada para Swagger.

```
class UserBase(BaseModel): username: str  
  
class UserCreate(UserBase): password: str
```

Versión Refactorizada (Después): La nueva implementación utiliza Pydantic de forma estricta. Se incorporan tipos de datos específicos (userName), restricciones de longitud (min_length) y metadatos (description, example). Esto genera automáticamente documentación interactiva y previene la inyección de datos "basura" en la capa de entrada.

```
class UserCreate(BaseModel): username: userName = Field(  
    ...,  
    title="Username",  
    description="Nombre único de usuario ",  
    example="usuariol")  
password: str = Field(  
    ...,  
    min_length=8,  
    title="Password",  
    description="Contraseña segura para autenticación"  
)
```

Anexo D: Evidencia de Análisis Estático y Dinámico

Este anexo compila la evidencia empírica generada por las herramientas de aseguramiento de calidad (SQA), contrastando el estado inicial del sistema (diagnóstico) con el estado final tras la ejecución del plan de mejora.

Los reportes certifican cuantitativamente la reducción de la deuda técnica, el aumento de la granularidad modular y el endurecimiento de la seguridad, validando el cumplimiento de los objetivos de calidad ISO/IEC 25010:2023.

Mantenibilidad y Deuda Técnica (SonarQube)

Las vistas generales (Overview) ilustran la evolución de la salud del código, destacando la transición de un estado crítico a uno saludable.



Estado Inicial (Diagnóstico): El análisis de línea base revela una calificación de fiabilidad "E" (Bugs), presencia de vulnerabilidades y una deuda técnica temporal acumulada significativa, con cobertura de pruebas nula.

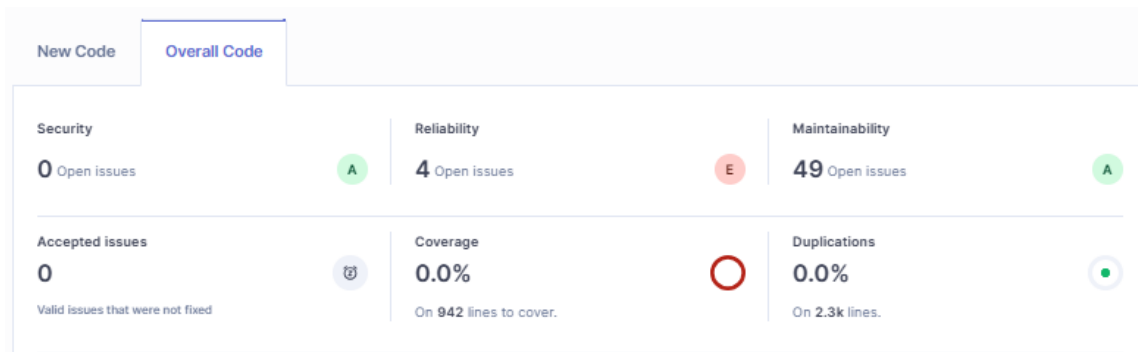


Ilustración D 1: Panel de control inicial evidenciando alta deuda técnica y baja fiabilidad

Fuente: Elaboración propia a través de análisis de SonarQube

Estado Final (Post-Intervención): Tras la refactorización, el sistema alcanza la calificación "A" en todas las métricas (fiabilidad, seguridad, mantenibilidad), logrando la amortización total de la deuda técnica de código y una reducción del 28% en la complejidad cognitiva.

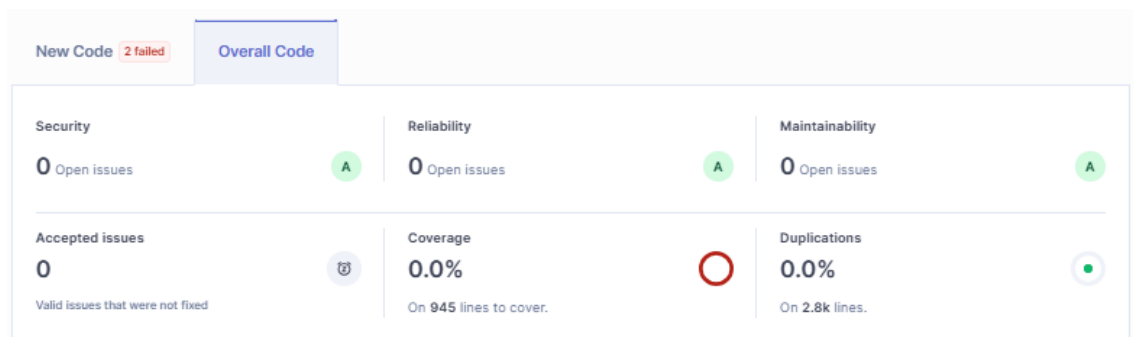


Ilustración D 2: Panel de control final certificando el saneamiento del código

Fuente: Elaboración Propia a través de análisis de SonarQube

Análisis Estructural (Emerge-viz)

Esta sección visualiza la transformación arquitectónica mediante grafos de dependencia. Los resultados demuestran un aumento positivo en la **granularidad**, evidenciando cómo la lógica concentrada en pocos archivos grandes (monolito) fue segregada en múltiples módulos cohesivos y específicos.

Caso 1: API Backend (Descomposición del Monolito): Se observa la transición desde una estructura con pocos nodos densos hacia una red distribuida de componentes pequeños (controladores, servicios, esquemas), facilitando la *testeabilidad* y el mantenimiento aislado.



Ilustración D 3: Estructura original de la API (Baja granularidad, alta concentración de lógica)

Fuente: Elaboración propia mediante Emerge-viz

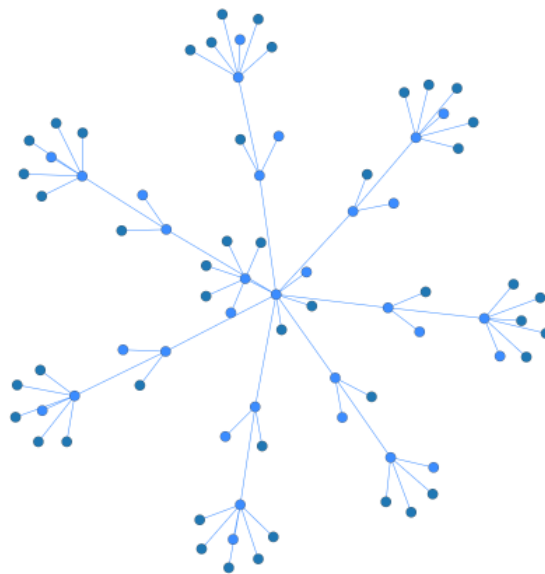


Ilustración D 4: Estructura refactorizada mostrando mayor dispersión modular y especialización (Alta granularidad)

Fuente: Elaboración propia mediante Emerge-viz



Caso 2: Servicio de IA (Normalización) La visualización post-refactorización refleja la extracción de utilidades, configuraciones y lógica de conexión a bases de datos hacia módulos independientes, reduciendo el acoplamiento en el script principal de inferencia.



Ilustración D 5: Estructura inicial del servicio de IA

Fuente: Elaboración propia mediante Emerge-viz



Ilustración D 6: Estructura final con segregación de responsabilidades

Fuente: Elaboración propia mediante Emerge-viz

Seguridad Perimetral (OWASP ZAP)

El análisis dinámico (DAST) sobre la API en ejecución permite visualizar la efectividad de las medidas de *hardening* aplicadas en la capa de transporte y configuración del servidor.



Estado Inicial (Vulnerabilidades Activas): El escaneo preliminar detectó alertas de riesgo medio relacionadas con la permisividad de CORS y la ausencia de cabeceras de seguridad HTTP, exponiendo la API a vectores de ataque web comunes.

Generated on Sun, 15 Jun 2025 23:09:50

ZAP Version: 2.16.1

ZAP by [Checkmarx](#)

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	6
Low	4
Informational	3
False Positives:	0

Ilustración D 7: Reporte inicial mostrando vulnerabilidades de configuración y protección de transporte deficientes

Fuente: Elaboración propia mediante OWASP ZAP

Estado Final (Mitigación Completa): Tras la implementación de políticas restrictivas y cabeceras de seguridad, el reporte final certifica la eliminación de alertas de riesgo alto y medio, validando un estado de seguridad robusto.

Generated on Sun, 30 Nov 2025 05:25:24

ZAP Version: 2.16.1

ZAP by [Checkmarx](#)

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	1
Informational	0
False Positives:	0

Ilustración D 8: Reporte de escaneo dinámico final confirmando la mitigación de riesgos de seguridad web

Fuente: Elaboración propia mediante OWASP ZAP



Consolidado de Hardening y Estándares (SAST)

Para las herramientas de análisis estático de seguridad y estilo (cuyos reportes son extensos en texto), se presenta la Tabla D 1 que sintetiza los hallazgos críticos previos y posteriores a la intervención.

Tabla D 1: Resumen de Hallazgos por Herramienta Específica

Fuente: Elaboración propia

Herramienta	Ámbito de Análisis	Hallazgos Críticos (Antes)	Hallazgos Críticos (Después)
Bandit	Seguridad (Python)	1 (Timeout ausente)	0
Dependency-Check	Vulnerabilidades (SCA)	1 (CVE en PyTorch)	0
Pylint	Calidad y Estilo (PEP-8)	Puntaje: 5.56/10	Puntaje: >8.0/10

Anexo E: Protocolo y Evidencia de Validación Funcional (Pruebas de Regresión y Flujos Críticos)

Dada la ausencia inicial de cobertura de pruebas automatizadas identificada en el diagnóstico (0% de cobertura), se formalizó un protocolo de pruebas de regresión manuales (Critical Path Testing) para mitigar el riesgo de regresiones.

Definición del Flujo Crítico

La siguiente tabla detalla la batería de pruebas secuencial diseñada para validar el ciclo de vida completo de una detección, abarcando desde el registro de usuario hasta la persistencia de resultados procesados por la IA.

Tabla E 1: Matriz de Validación de Flujo Crítico

Elaboración propia basada en colección de pruebas de Postman

ID	Caso de Prueba	Endpoint / Método	Datos de Entrada (Input)	Resultado Esperado (Criterio de Aceptación)
ST-01	Registro de Usuario	POST /users/	Body (JSON): { "username": "userTester", "password": "..." }	Código 200 OK. El sistema crea el recurso y retorna los datos del usuario sin exponer el hash de la contraseña.
ST-02	Autenticación (Login)	POST /auth/token	Form-Data: username="userTester", password="..."	Código 200 OK. La respuesta incluye access_token (JWT) válido y token_type: bearer.
ST-03	Ingesta de Imagen	POST /images/upload	Header: Authorization: Bearer <token>; Body: Archivo .jpg vía multipart/form-data.	Código 200 OK. La API confirma la recepción y el encolamiento en RabbitMQ sin bloquear el hilo principal.
ST-04	Consulta de Resultados	GET /detections/{id}	Header: Authorization: Bearer <token>; Path: ID retornado en ST-03.	Código 200 OK. El JSON de respuesta incluye coordenadas y la lista de etiquetas normalizadas (ej: "bache").

Evidencia de Ejecución

A continuación, se presentan las evidencias gráficas de la ejecución exitosa de los pasos más relevantes del protocolo, demostrando la integridad de la seguridad, la capacidad asíncrona y la consistencia de los datos, todo esto usando Postman[28].

1. Validación de Seguridad y Autenticación (ST-02) La Ilustración E 1 evidencia la correcta emisión de credenciales seguras (JWT), confirmando que el sistema de gestión de usuarios refactorizado opera correctamente.

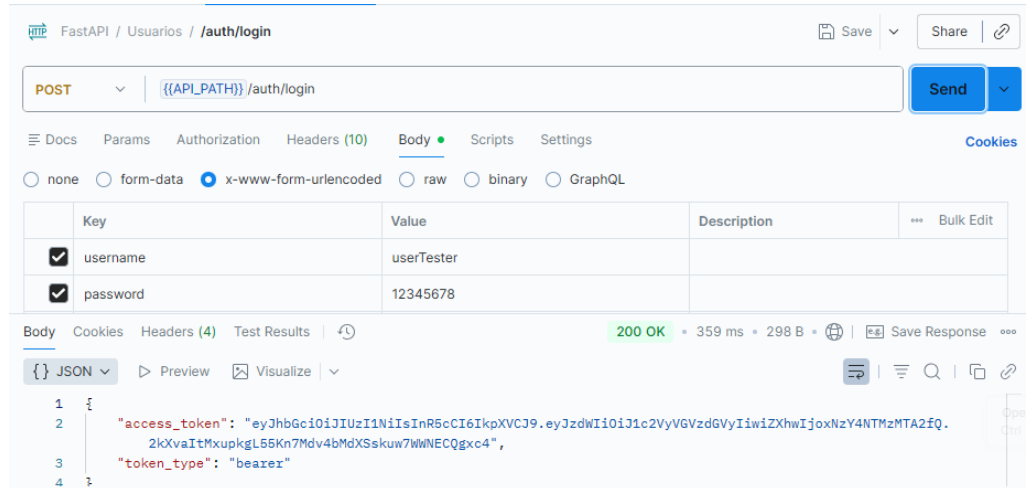


Ilustración E 1: Respuesta exitosa del endpoint de autenticación generando un token de acceso

Fuente: Elaboración propia mediante Postman

2. Validación de Ingesta Asíncrona (ST-03) La Ilustración E 2 demuestra la capacidad de la API para manejar cargas de trabajo pesadas sin bloqueo. El código de estado 200 OK confirma que la imagen fue recibida y delegada exitosamente a la cola de mensajería (RabbitMQ) para su procesamiento diferido.

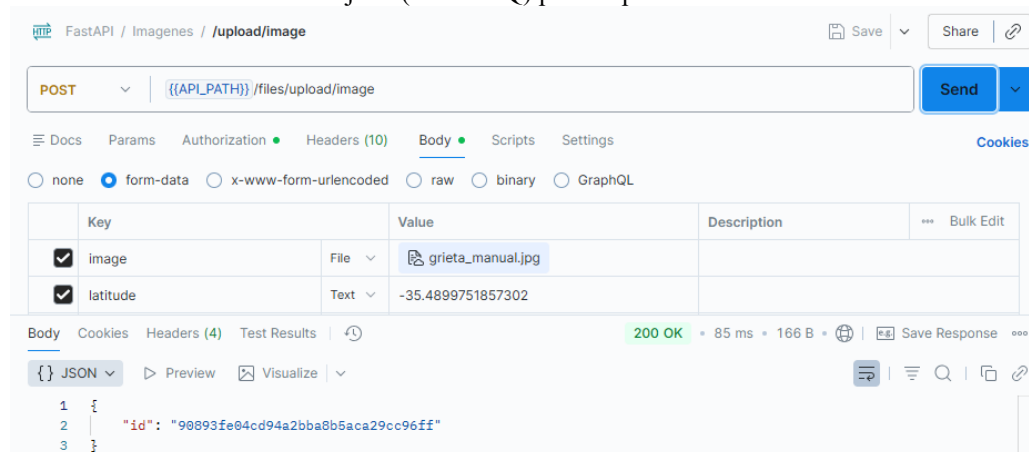


Ilustración E 2: Respuesta de aceptación inmediata del servidor, validando el desacoplamiento entre recepción y procesamiento

Fuente: Elaboración propia mediante Postman



3. Validación de Integración y Contrato de Datos (ST-04) Finalmente, la Ilustración E 3 confirma que el flujo completo (API → RabbitMQ → Worker IA → MongoDB) concluyó exitosamente. Se observa que el JSON de respuesta cumple con el esquema estricto definido, entregando la clasificación de la irregularidad.

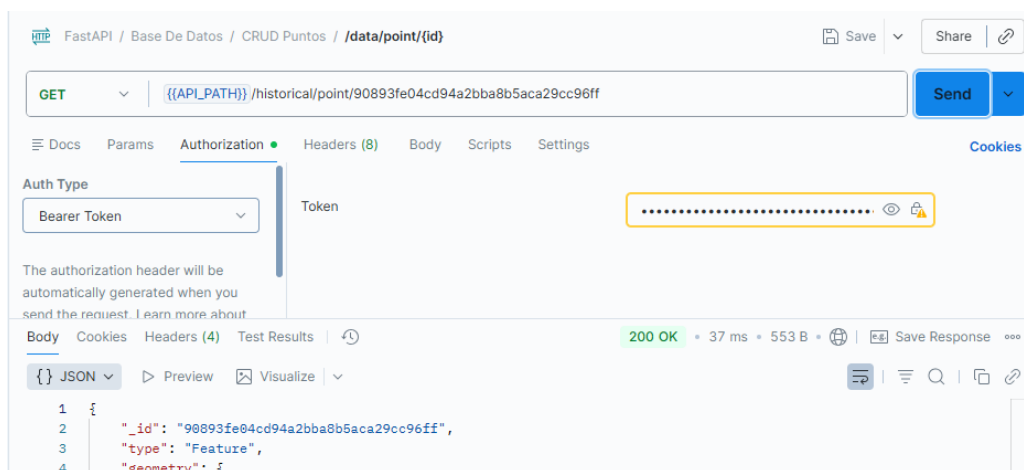


Ilustración E 3: Respuesta final del sistema mostrando la detección persistida y formateada según el nuevo contrato

Fuente: Elaboración propia mediante Postman