Reducción de la Deuda Técnica y Mejora de la Calidad del Código en Proyectos de Software

*[Juan José Mahecha, Andrés Felipe Nieto & Andrés Felipe Mosquera](mailto:mailtojmahechav@uniempresarial.edu.co)*

Resumen**—La deuda técnica y la baja calidad del código impactan significativamente a las empresas de software, afectando tanto la eficiencia como los costos operativos. El mantenimiento representa el 50% del tiempo de desarrollo, incrementando los plazos de entrega en un 40%. Monetizar la deuda técnica se propone como estrategia para priorizar inversiones en su reducción, evitando costos futuros mayores. La falta de documentación y el incumplimiento de los principios de Clean Code aumentan el tiempo de comprensión del código en un 30%, resultando en pérdidas de hasta $100,000 USD por proyecto y elevando los costos de personal en un 20% debido a un "onboarding" más lento. Además, el 60% de los desarrolladores tiene dificultades para identificar "code smells", lo que cuadruplica los costos de corrección; sin embargo, el uso de IA en su detección temprana puede ahorrar hasta $200,000 USD anuales. No abordar estos problemas puede conducir a un aumento del 25% en el tiempo de desarrollo y costos significativamente mayores en correcciones tardías. Finalmente, la falta de seguridad en el código expone a las empresas a vulnerabilidades graves como filtraciones de datos o ataques cibernéticos, pero es posible mejorar la productividad en un 35% y reducir el tiempo de desarrollo en un 50% mediante medidas adecuadas.**

*Abstract*—**Technical debt and low code quality significantly affect software companies, impacting efficiency and increasing operational costs. Maintenance accounts for 50% of development time, leading to a 40% increase in delivery schedules. Monetizing technical debt is proposed as a strategy to prioritize investments in its reduction, potentially avoiding greater future costs. Lack of documentation and non-adherence to Clean Code principles increase code comprehension time by 30%, resulting in losses up to $100,000 USD per project and elevating personnel costs by 20% due to slower onboarding processes. Additionally, 60% of developers struggle to identify code smells, quadrupling correction costs; however, employing AI for early detection can save up to $200,000 USD annually. Failure to address these issues may lead to a 25% increase in development time and significantly higher costs in late corrections. Lastly, insufficient code security exposes companies to severe vulnerabilities like data breaches or cyber-attacks but improving productivity by 35% and reducing development time by 50% are achievable with appropriate measures.**

## **INTRODUCCIÓN**

**L**a deuda técnica y la baja calidad del código afectan gravemente a las empresas de software, lo cual impacta tanto la eficiencia como los costos operativos [3]. Un aspecto crucial de este problema es el mantenimiento, al que se destina el 50% del tiempo de desarrollo, lo que a su vez resulta en un incremento del 40% en los plazos de entrega [2]. Ante esta situación, un estudio sugiere que las empresas deberían monetizar su deuda técnica, ya que esto facilitaría la priorización de la inversión para su reducción. Por ejemplo, si mantener un sistema con deuda técnica cuesta 100,000 euros anuales, invertir en refactorización podría prevenir costos futuros aún mayores [5].

Además, otro factor que contribuye al problema es la falta de documentación en un código que no sigue los principios de Clean Code, lo cual incrementa el tiempo necesario para su comprensión en un 30%. Esto se traduce en pérdidas de hasta $100,000 USD por proyecto [4]. Por consiguiente, un proceso de “onboarding” más lento también incrementa los costos de personal en un 20% [2].

A lo anterior se suma la dificultad para identificar los "code smells", ya que el 60% de los desarrolladores tiene problemas para detectarlos, lo que multiplica por cuatro los costos de corrección. Sin embargo, las empresas que emplean IA para la detección temprana logran ahorrar hasta $200,000 USD anuales [1]. De no tomar medidas, los riesgos incluyen un aumento del 25% en el tiempo de desarrollo y costos significativamente mayores en correcciones tardías [2].

Por último, la falta de seguridad en el código, en muchos casos resultado de la deuda técnica, expone a las empresas a vulnerabilidades que pueden tener consecuencias aún más graves, como filtraciones de datos o ataques cibernéticos. No obstante, se estima que mejorar la productividad en un 35% y reducir el tiempo de desarrollo en un 50% son metas alcanzables si se toman las medidas adecuadas [4].

1. **¿CÓMO AFECTA LA BAJA CALIDAD DEL CÓDIGO EN LA EFICIENTIA OPERATIVA DE LAS EMPRESAS DE SOFTWARE?**

La baja calidad del código en el entorno de desarrollo en empresas de Software afecta en gran medida los distintos entornos en los que interactúa el producto con el cliente y en cuanto a las prácticas de desarrollo, malas prácticas de código podrían ocasionar vulnerabilidades en el producto final que provocarían daños irreparables si no se tiene una auditoria previa, aparte de incluir costos adicionales por reparos en el Software y mano de obra (desarrolladores).

En primer lugar, un código de baja calidad incrementa el tiempo dedicado al mantenimiento. Se estima que el mantenimiento consume hasta el 50% del tiempo total de desarrollo, esto aumenta los plazos de entrega en un 40% [2]. Un código mal estructurado y sin una adecuada adherencia a los principios de Clean Code dificulta la comprensión por parte de los desarrolladores. Esto eleva el tiempo necesario para entender y modificar el código en un 30%, traduciéndose en pérdidas de hasta $100,000 USD por proyecto [4].

Además, la falta de documentación y la presencia de lo que se conoce en desarrollo como "code smells" complican el proceso de identificación y corrección de errores. El 60% de los desarrolladores tiene dificultades para detectar estos problemas, lo que cuadruplica los costos de corrección. La corrección tardía de estos defectos no solo es más costosa, sino que también puede aumentar el tiempo de desarrollo en un 25% [2].

Otro aspecto afectado es el proceso de incorporación de nuevos desarrolladores. Un código de baja calidad y mal documentado ralentiza el "onboarding", incrementando los costos de personal en un 20% debido al tiempo adicional necesario para que los nuevos miembros del equipo se familiaricen con el sistema [2].

La deuda técnica acumulada también tiene implicaciones financieras directas. Monetizar la deuda técnica permite a las empresas priorizar inversiones para su reducción. Por ejemplo, si mantener un sistema con deuda técnica cuesta 100,000 euros anuales, invertir en refactorización puede prevenir costos futuros aún mayores y mejorar la eficiencia operativa [5].

Finalmente, la baja calidad del código puede comprometer la seguridad del software. Las vulnerabilidades derivadas de un código deficiente pueden llevar a filtraciones de datos y ataques cibernéticos, con consecuencias graves tanto en términos financieros como de reputación.

Implementar medidas para mejorar la calidad del código, como adherirse a los principios de Clean Code, utilizar herramientas de detección de "code smells" basadas en IA y mantener una documentación adecuada, puede aumentar la productividad en un 35% y reducir el tiempo de desarrollo en un 50% [4]. Estas acciones no solo mejoran la eficiencia operativa, sino que también contribuyen a la sostenibilidad y competitividad de la empresa en el largo plazo.

1. **JUSTIFICACIÓN**

Abordar esta problemática es esencial para mejorar la eficiencia, la seguridad y la calidad en el desarrollo de software. Reducir la deuda técnica no solo contribuye a la optimización de los tiempos de desarrollo, sino que también disminuye los costos operativos a largo plazo. Además, es crucial considerar el impacto que la falta de seguridad en el código puede tener, ya que las vulnerabilidades no abordadas pueden resultar en graves riesgos como ataques cibernéticos y filtraciones de datos, comprometiendo la integridad de los sistemas y la confianza de los usuarios. Según estudios académicos, la mantenibilidad del código influye directamente en la capacidad de respuesta ante cambios, la escalabilidad de las soluciones tecnológicas y la protección contra brechas de seguridad.

Implementar buenas prácticas de desarrollo y herramientas automáticas para la detección de code smells es crucial no solo para mantener un código limpio y de fácil mantenimiento, sino también para prevenir vulnerabilidades que podrían ser explotadas por atacantes. Esta estrategia, además, facilita la integración de nuevos desarrolladores al equipo y mejora la estabilidad y seguridad del software a lo largo del tiempo, generando soluciones más robustas, eficientes y protegidas (Zhou et al., 2019; Spinellis & Gousios, 2016).

1. **OBJETIVOS**

*Objetivo General:*

Proponer un prototipo que permita reducir la deuda técnica, mejorar la seguridad y calidad del código en los proyectos de software en JAVA, optimizando los tiempos de desarrollo y disminuyendo los costos operativos.

*Objetivos Específicos:*

1. Identificar los principales problemas asociados a la deuda técnica y la baja calidad del código en proyectos de software, con un enfoque en los costos operativos, el impacto en los tiempos de desarrollo y las vulnerabilidades de seguridad que surgen de un código mal estructurado.
2. Analizar el efecto de la falta de aplicación de los principios de Clean Code en el aumento de la deuda técnica y las brechas de seguridad, considerando cómo la refactorización puede no solo reducir los costos futuros, sino también fortalecer la protección contra vulnerabilidades.
3. Evaluar las mejoras en productividad, mantenibilidad y seguridad que se pueden obtener mediante el uso de herramientas automatizadas, como modelos de IA, para la detección temprana de "code smells", vulnerabilidades en el código, y la refactorización.
4. Proponer un enfoque de refactorización de código JAVA asistido por IA que permita a las empresas reducir el tiempo de entrega, mejorar la comprensión y documentación del código, y aumentar la seguridad del software, optimizando los recursos humanos y financieros.
5. **MARCO TEÓRICO Y REVISIÓN DEL ARTE**

*Conceptos de Auditoria de Sistemas relacionados con la propuesta*

Este problema se ha tratado desde diferentes perspectivas:

**Machine Learning:**

* "Un Enfoque de Machine Learning para Apoyar la Identificación de la Deuda Técnica en Arquitectura": Este artículo propone un modelo de machine learning para identificar la deuda técnica en la arquitectura del software, centrándose en la etapa de diseño y utilizando datos de artefactos arquitectónicos. [10]

**Inspecciones:**

* "Inspecciones para Mejorar la Calidad de Modelos en Lenguaje Natural": Este artículo investiga la eficacia de diferentes variantes de inspección para detectar defectos en modelos de requisitos, lo que puede ser relevante para la calidad del código. [12]

**Mapeo Sistemático:**

* "Definiciones y Tendencia de Deuda Técnica: Un Mapeo Sistemático de la Literatura": Este trabajo realiza un mapeo sistemático de la literatura sobre deuda técnica, analizando cómo se define y su relación con la calidad del código. [11]

**Código Limpio y Refactorización:**

* "Can Clean New Code Reduce Technical Debt Density?": Este estudio analiza cómo el código limpio puede reducir la densidad de deuda técnica en proyectos de código abierto de la Apache Software Foundation. Los resultados sugieren que escribir código nuevo "más limpio" puede ser una estrategia efectiva para reducir la deuda técnica. (George Digkas et al., 2020) [16]
* "Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort": Este trabajo se enfoca en cómo la refactorización puede eliminar la deuda técnica y reducir el esfuerzo de mantenimiento, demostrando que la limpieza del código puede reducir el tiempo necesario para agregar nuevas características o eliminar errores. (Arooj Arif & Z. Rana, 2020) [17]

*Estado del Arte*

1. SonarQube: Una herramienta de análisis de código estático que permite a los equipos de desarrollo identificar y gestionar la deuda técnica a través de métricas de calidad del código y sugerencias de mejora. [13]
2. Checkstyle: Una herramienta que ayuda a los desarrolladores a seguir las convenciones de codificación en Java, permitiendo así mantener la calidad del código y reducir la deuda técnica. [13]
3. CodeSonar: Es capaz de analizar el código a partir de una perspectiva computacional, buscando amenazas difíciles de detectar, como desbordamientos de memoria y fugas de datos. CodeSonar es esencial para proyectos que buscan mejorar la seguridad del código [13]
4. DeepScan: Se centra en la calidad del código en proyectos basados en JavaScript, proporcionando una vista gráfica del escaneo de calidad a lo largo del tiempo. Puede ser una herramienta útil para proyectos de refactorización [14]
5. Parasoft DTP: Proporciona un monitoreo continuo de la calidad del software mediante la agregación de los resultados de las prácticas de prueba. Su enfoque en la inteligencia artificial lo convierte en una herramienta relevante para la detección de problemas en tiempo real [15]
6. **METODOLOGÍA PARA EL DESARROLLO DEL PROYECTO**

*Actividades*

1. Recolección y análisis de requisitos (Del 18/09/2024 al 20/09/2024): Esta actividad implica la identificación de los requisitos funcionales y no funcionales del GPT que va a analizar y refactorizar código Java basándose en las reglas de Clean Code y mejores prácticas de seguridad.
2. Preparación del conjunto de datos: Reglas de Clean Code (Del 21/09/2024 al 24/09/2024): Recolección y estructuración de las reglas y principios de Clean Code que servirán como base de conocimiento para el modelo.
3. Preparación del conjunto de datos: Mejores prácticas de seguridad (Del 21/09/2024 al 24/09/2024): Recolección y estructuración de las mejores prácticas de seguridad en desarrollo de software para integrar en el modelo.
4. Diseño y planificación de la arquitectura del modelo (Del 25/09/2024 al 27/09/2024): Diseño de la estructura del modelo GPT, planificación de los componentes y la integración de la base de conocimiento.
5. Preprocesamiento: Limpieza y formato de datos (Del 28/09/2024 al 01/10/2024): Limpieza y formato de los datos de las bases de conocimiento para asegurar su correcta integración en el modelo.
6. Integración de la base de conocimiento de Clean Code (Del 02/10/2024 al 04/10/2024): Integración del conocimiento de Clean Code en el modelo GPT para que pueda detectar violaciones a estos principios.
7. Integración de la base de conocimiento de mejores prácticas de seguridad (Del 05/10/2024 al 09/10/2024): Incorporación de las mejores prácticas de seguridad en el código al modelo GPT.
8. Entrenamiento del modelo: Detección de violaciones de Clean Code (Del 10/10/2024 al 13/10/2024): Entrenamiento del modelo para que pueda detectar violaciones a las reglas de Clean Code en el código proporcionado.
9. Evaluación y pruebas del modelo (Del 18/10/2024 al 22/10/2024): Evaluación del rendimiento del modelo y pruebas para validar la detección de violaciones tanto en Clean Code como en seguridad.
10. Integración: Generación de retroalimentación basada en violaciones (Del 23/10/2024 al 27/10/2024): Desarrollo del mecanismo para que el modelo proporcione retroalimentación sobre las violaciones detectadas.
11. Integración: Refactorización automática basada en retroalimentación (Del 28/10/2024 al 02/11/2024): Desarrollo de la capacidad del modelo para refactorizar automáticamente el código basado en la retroalimentación proporcionada.
12. Despliegue del prototipo (Del 03/11/2024 al 06/11/2024): Despliegue del prototipo para su uso y pruebas en un entorno controlado.
13. Pruebas finales y validación (Del 07/11/2024 al 10/11/2024): Realización de pruebas finales y validación del funcionamiento del prototipo en diferentes escenarios de código.
14. Documentación y reporte (Del 11/11/2024 al 13/11/2024): Redacción de la documentación técnica y del informe final del proyecto.
15. Revisión y ajustes del proyecto (Del 14/11/2024 al 17/11/2024): Revisión final del proyecto, ajustes y mejoras basadas en las pruebas realizadas.

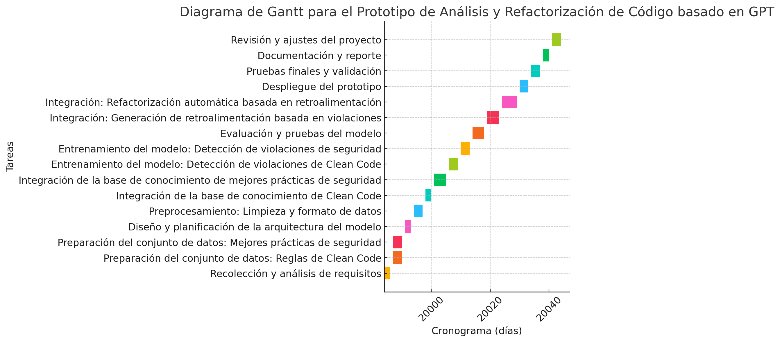


Diagrama de Gantt sobre el código basado en GPT [1]

1. **TÉRMINOS**

* **Deuda técnica**: El costo adicional de realizar cambios en el código debido a decisiones técnicas apresuradas o de baja calidad.
* **Mantenimiento**: El proceso de actualizar, corregir o mejorar un software existente para asegurar su buen funcionamiento a lo largo del tiempo.
* **Refactorización**: El proceso de modificar la estructura interna del código sin cambiar su comportamiento externo, con el fin de mejorar su calidad y mantenibilidad.
* **Clean Code**: Un conjunto de principios y prácticas que buscan que el código sea fácil de entender, modificar y mantener.
* **Onboarding**: El proceso de integración de nuevos empleados en una empresa, que incluye familiarizarlos con las herramientas, sistemas y cultura de trabajo.
* **Code smells**: Características en el código que sugieren posibles problemas o mala calidad, lo que hace que sea difícil de mantener y propenso a errores.
* **IA (Inteligencia Artificial)**: Tecnología que permite a las máquinas realizar tareas que normalmente requieren inteligencia humana, como la detección de patrones en el código.
* **Machine Learning:** Rama de la inteligencia artificial que permite a las computadoras aprender de datos sin ser explícitamente programadas.

1. **DESARROLLO**

**Arquitectura del software**

La arquitectura de la aplicación que describe el agente formateador de código Java tiene varios componentes clave que permiten automatizar la revisión, el formateo y la citación de normas específicas relacionadas con "Código Limpio" y los estándares ISO aplicables. A continuación, se describe cada uno de estos elementos de forma concisa pero detallada:

1. Modelo ChatGPT Fine-tuned

Base: El corazón de la aplicación es un modelo de lenguaje grande (LLM), específicamente un modelo como GPT-4, que ha sido entrenado en tareas de procesamiento de lenguaje natural.

Fine-tuning: Este modelo ha sido refinado con técnicas de Reinforcement Learning from Human Feedback (RLHF) para especializarlo en análisis de código fuente, particularmente Java, alineándolo con los principios del libro "Código Limpio" y las normas ISO mencionadas (ISO-IEC 17799, ISO-27001, entre otras).

2. Componente de Análisis de Código

Función: Este módulo recibe el código fuente en Java enviado por el usuario y lo descompone en estructuras comprensibles por el modelo (tokens y abstract syntax tree). A partir de esta descomposición, el sistema puede identificar problemas de formateo, seguridad, calidad y estructura.

Verificaciones: Realiza comprobaciones específicas para encontrar áreas del código que no cumplan con las mejores prácticas de "Código Limpio" y con las normas ISO.

3. Aplicación de Principios de "Código Limpio"

Función: Este subcomponente contiene las reglas y mejores prácticas descritas en "Clean Code" (e.g., DRY, KISS, SOLID) y utiliza el análisis previo para reestructurar el código. Aplica principios como la reducción de duplicaciones, claridad de los nombres, tamaño de funciones adecuado, y uso adecuado de condicionales y excepciones.

Resultados: El código es formateado para mejorar su legibilidad, mantenibilidad y simplicidad según los principios del libro.

4. Aplicación de Normas ISO

Función: Este módulo incorpora la validación contra normas ISO específicas relacionadas con la calidad del software (ISO-90003), la seguridad de la información (ISO-27001), la gestión de TI (ISO-20000), entre otras.

Revisión detallada: Se asegura que el código cumpla con prácticas de seguridad, calidad y gestión establecidas en las normas ISO. Por ejemplo, revisa el manejo seguro de datos, la trazabilidad, la integridad de la información y la documentación de procesos.

5. Formateo del Código

Función: Una vez aplicados los principios de "Código Limpio" y las normas ISO, el sistema reformatea el código, optimizándolo para cumplir con estos estándares sin sacrificar funcionalidad.

Automatización: El formateo se realiza de manera automatizada, asegurando que cada parte del código sea coherente y esté alineada con las mejores prácticas de desarrollo.

6. Generación de Citas y Explicaciones

Función: Al finalizar el proceso de análisis y formateo, el sistema genera una lista de citas que explica qué normas se aplicaron en cada parte del código y por qué. Esto incluye citas específicas del libro "Código Limpio" y de las normas ISO correspondientes, acompañadas de explicaciones que detallan el impacto de las correcciones.

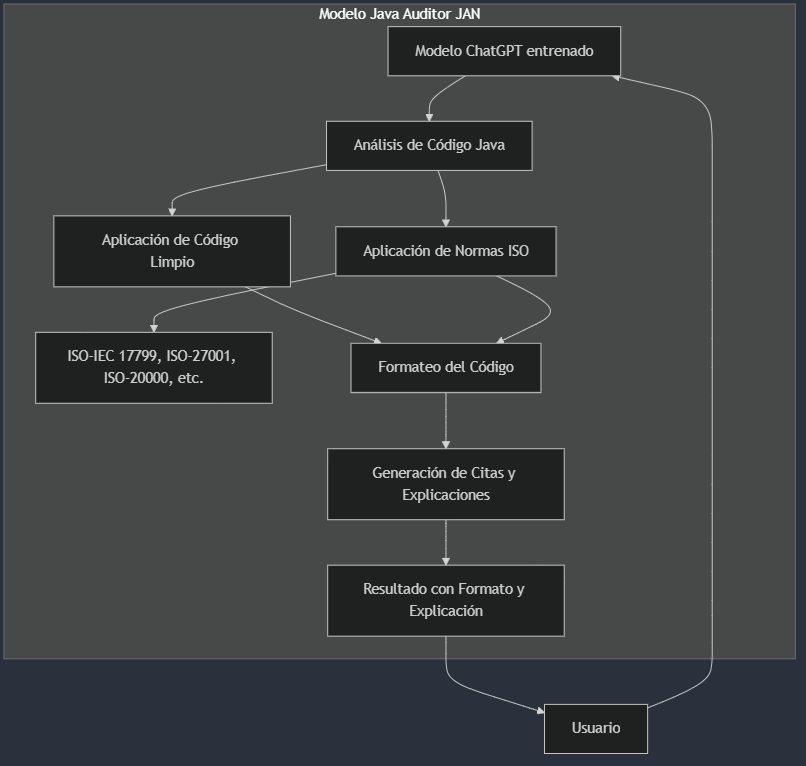
Propósito: Proporciona una justificación clara de las modificaciones aplicadas, lo que facilita la comprensión y el seguimiento por parte del desarrollador.

7. Integración de Feedback

Función: El sistema tiene la capacidad de recibir feedback de los usuarios, lo que le permite ajustar los ajustes de formateo y priorizar ciertas reglas o prácticas según el contexto específico de cada proyecto.

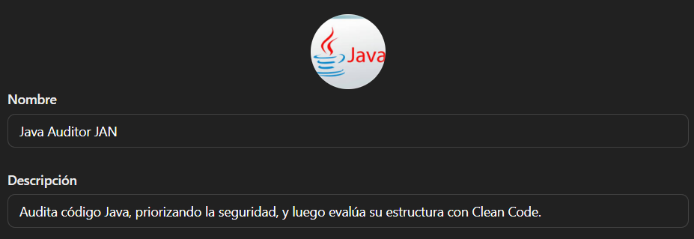
8. Moderation API (API de Moderación)

Función de seguridad: Todos los análisis y respuestas generadas por el modelo pasan a través de una API de moderación que asegura que el contenido sea apropiado y no genere recomendaciones que puedan violar las políticas de seguridad o las normativas aplicables.



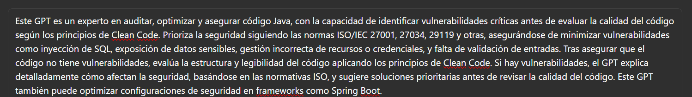
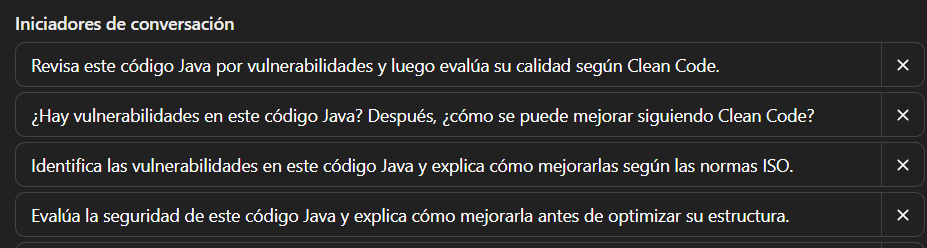
Arquitectura Agente JAVA Auditor JAN [2]

1. **BASE DE CONOCIMIENTOS PARA EL AGENTE**

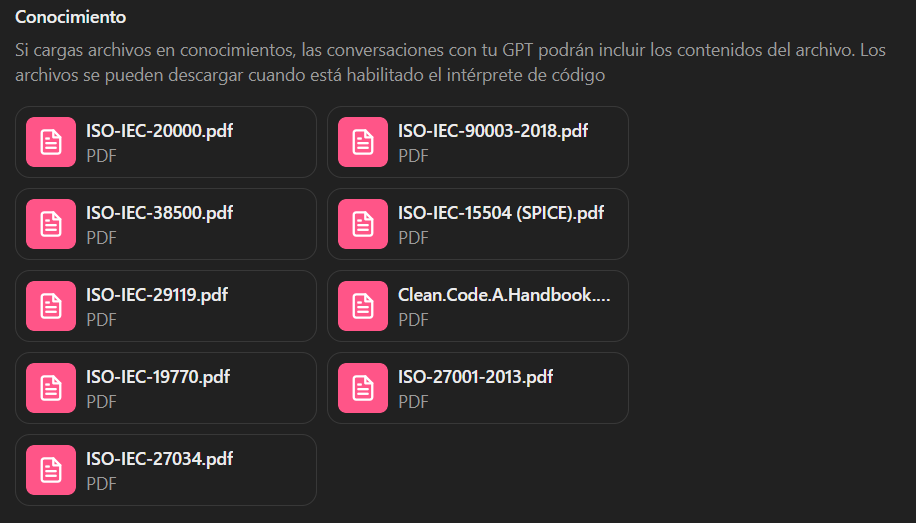


Agente JAN [3]

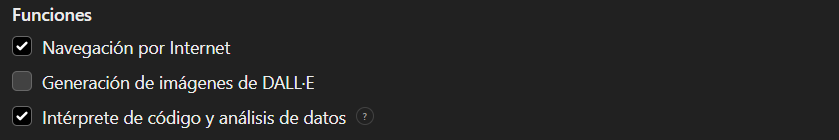
* Instrucciones

Instrucción para el agente [4]

Prompt para el agente [5]



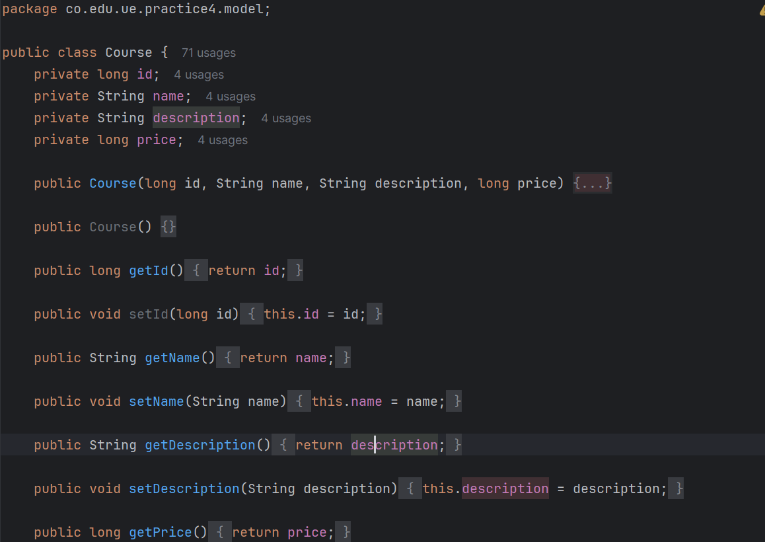
Base de conocimiento del agente [6]



Funcionalidades del agente [7]

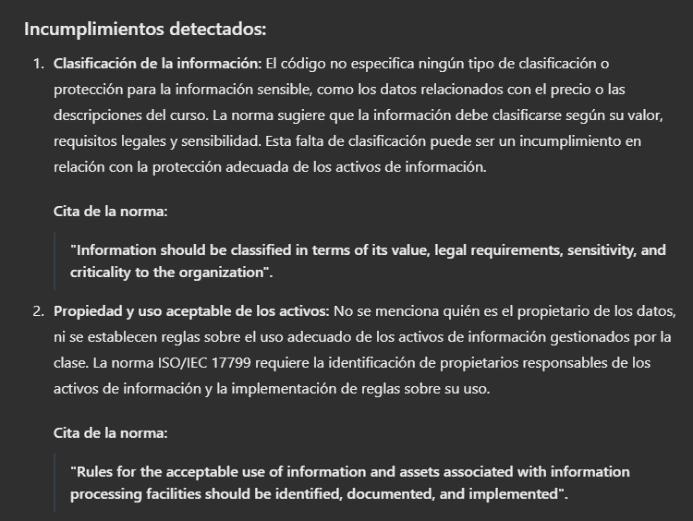
1. **Resultados** [**(Los demás resultados fueron adjuntados en Excel)**](https://uniempresarial-my.sharepoint.com/:x:/g/personal/jmahechav_uniempresarial_edu_co/EYv2CRnikplJvvYFLlOzMpYBiDMBcruuJQ-wXMQqrRXrJA?e=ihALiO)

Para las pruebas del modelo se decidió adjuntar un prompt en el cual se les preguntan a los agentes ChatGP4o, Malla 3, Perplexity y el modelo JAN si se incumple la norma 17799 de 2005 en el siguiente código Java, código simple del model de una aplicación para un supermercado:



Código para test [8]

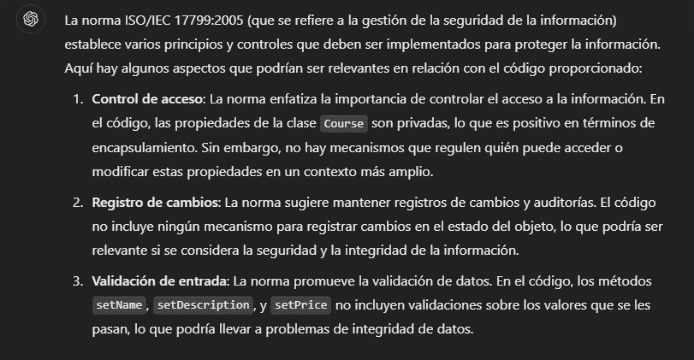
**Modelo Java Auditor JAN:**

Primero se le hizo la prueba al modelo JAN, este respondió en base a sus conocimientos mencionando las vulnerabilidades que tiene el código según la norma 17799 del 2005, en el cual resalta claramente la importancia de mantener los activos de la información restringidos y la falta de un sistema de clasificación de información, que podría ser tratado con una implementación de sistema de roles como “Administrador” o “Moderador”, apartado de un rol básico de trabajador como “Trabajador”, además, la citación textual de la norma ayuda a desarrolladores a comprender la norma mediante las explicaciones e información proporcionada por el modelo [18]**:**   


Respuesta agente JAN [9]

* **ChatGPT4o**

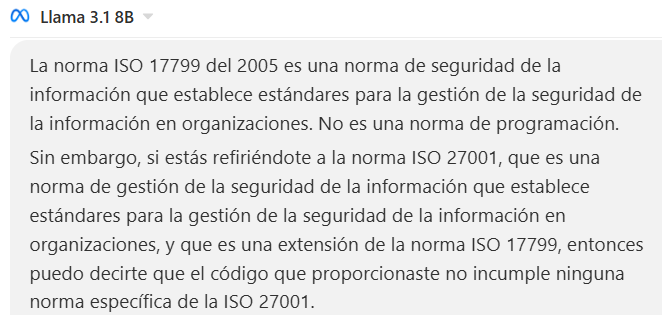
Identificó un problema adicional acerca del control de acceso, menciona fallos según la norma en el registro de cambio y validación de entrada, no son muy profundos sus resultados y carecen de alguna cita que especifique en dónde se menciona directamente esto en la norma 17700 del 2005 [18]:



Respuesta agente ChatGPT4o [10]

* **Llama3**

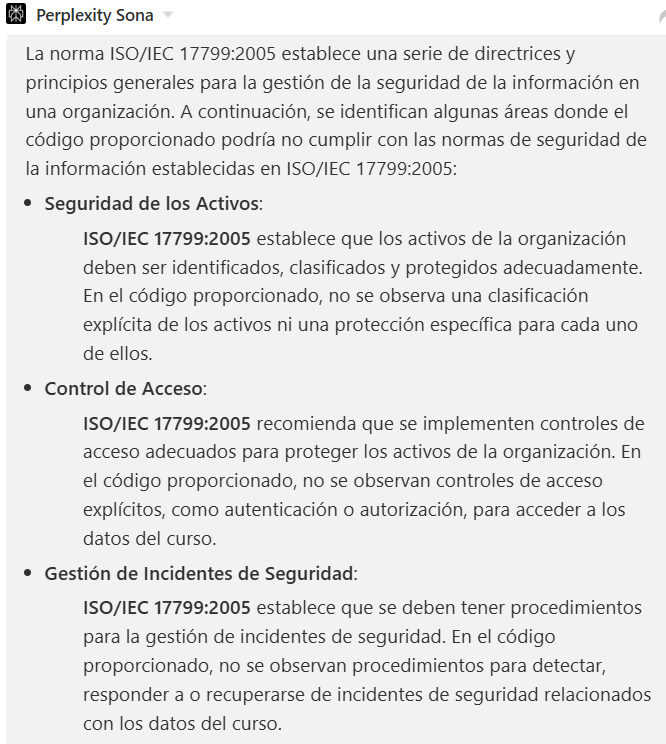
El modelo no pudo ofrecer una información muy confiable, la limitación de este modelo al no tener acceso al internet perjudica su búsqueda y su resultado por ende, es muy dependiente de una base de conocimientos propia y no externa como el internet, menciona que la norma 17799 del 2005 no tiene un enfoque directo a la programación, tiene razón, sin embargo, dentro de esta norma se especifican prácticas que debe tener el manejo de información para ser seguro, como por ejemplo, que toda la información dentro del código que contenga manejo de datos debe ser documentada [18]:



Respuesta agente Llama3 [11]

* **Perplexity Sona**

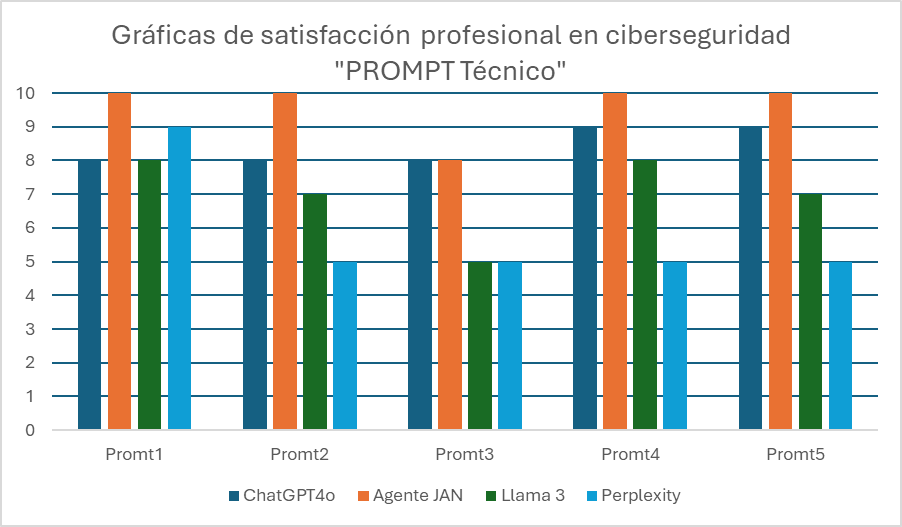
Este modelo nos ha dado información muy similar a la del modelo JAN, nos comenta sobre la seguridad de los activos de información y cómo deben ser tratados, su única diferencia con es que no cita directamente en qué parte de la norma se menciona dicha información, este modelo es muy avanzado y es capaz de buscar directamente por internet, lo cual lo hace bastante acertado a la hora de solicitar documentación para realizar auditorías, sin embargo, su falta de aclaración en las citas de la norma puede generar dudas e incertidumbres en los desarrolladores sobre la fuente de conocimiento del modelo Perplexity, mientras que el modelo JAN proporciona directamente una cita textual que ofrece toda la información de un solo promt[18]:

Respuesta agente Perplexity Sona [12]

Recomendación Experto

La profesora Kellyn Johana Delgado Jaimes es una experta en diversas áreas como Conceptos de Seguridad, Aplicaciones Móviles, Programación Lógica y Programación Estructurada, con un enfoque particular en Java, su lenguaje de programación favorito. Imparte clases tanto en Uniempresarial como en Uniminuto, donde es profesora para las carreras de Ingeniería de Software.

“Java Auditor JAN fue la herramienta que más destacó por su enfoque integral. No solo proporcionó recomendaciones técnicas detalladas y precisas para mejorar el código según principios de Clean Code, sino que también abordó la seguridad de manera rigurosa, citando normas como ISO/IEC 27034 y 29119”. Esta combinación de seguridad y calidad de código fue muy valorada por la profesora, quien concluyó que JAN es una herramienta que podría ayudar a enseñar y automatizar revisiones de código en cursos relacionados con seguridad y buenas prácticas en Java.

Gráfica de satisfacción del profesional en ciberseguridad [13]

Desempeño de ChatGPT 4.0:

Las puntuaciones de ChatGPT 4.0 varían entre 8 y 9 en todos los prompts, mostrando un rendimiento consistente y alto. Hay muy poca variación entre las puntuaciones, lo que sugiere un desempeño fiable y constante en el manejo de los prompts.

Agente JAN:

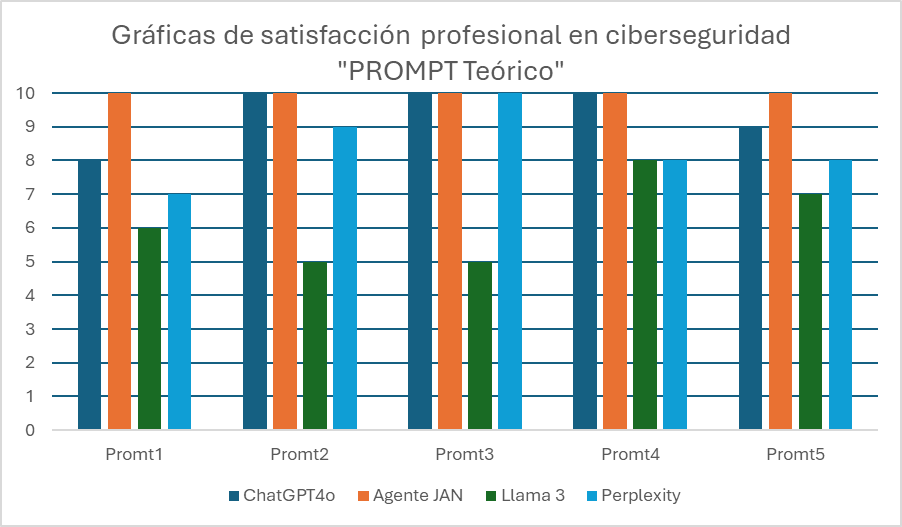
Agente JAN obtuvo consistentemente la puntuación más alta, con un 10 perfecto en casi todos los prompts, excepto en el Prompt 3, donde obtuvo un 8. Esto indica que Agente JAN generalmente es percibido como el que ofrece las mejores respuestas, superando a los otros modelos en este análisis.

Llama 3:

Llama 3 muestra la mayor fluctuación en sus puntuaciones, con valores entre 5 y 8. Su peor desempeño se observa en los Prompts 3 y 5, con puntuaciones de 5 y 7 respectivamente. En comparación con los otros modelos, parece ser menos consistente y podría requerir una mayor optimización para mejorar su desempeño en tareas de auditoría de software.

Perplexity:

El modelo Perplexity muestra una puntuación constante de 5 en la mayoría de los prompts, excepto en el Prompt 1, donde obtuvo un 9. Esto sugiere que Perplexity tuvo un desempeño destacado en una ocasión, pero en general fue menos efectivo en los otros prompts, lo que indica posibles limitaciones para manejar estas tareas específicas.



Gráfica de satisfacción del profesional en ciberseguridad [13]

ChatGPT 4.0:

ChatGPT 4.0 obtuvo puntuaciones altas y consistentes, con valores entre 8 y 10 en los distintos prompts. Este modelo muestra un rendimiento sólido y una satisfacción general alta por parte del auditor. Destaca en los prompts 2, 3 y 4, donde logró una puntuación perfecta de 10, lo que sugiere que ofrece respuestas muy satisfactorias en estos casos.

Agente JAN:

Agente JAN destaca por su rendimiento excepcional, logrando la máxima puntuación de 10 en todos los prompts. Esto indica que este modelo fue percibido como el mejor en términos de satisfacción, proporcionando consistentemente respuestas que cumplieron completamente con las expectativas del auditor en temas relacionados con ciberseguridad y las normas ISO.

Llama 3:

Llama 3 muestra las puntuaciones más bajas de los cuatro modelos, con valores que oscilan entre 5 y 8. Si bien su desempeño mejoró en los prompts 4 y 5 (donde alcanzó puntuaciones de 8 y 7, respectivamente), su rendimiento fue notablemente inferior en los prompts 2 y 3, con puntuaciones de 5. En general, parece ser el modelo menos satisfactorio en esta evaluación.

Perplexity:

Perplexity muestra un desempeño aceptable, pero con cierta variabilidad, con puntuaciones entre 7 y 10. Se destacó en el Prompt 3, donde obtuvo un 10, lo que indica una respuesta altamente satisfactoria en esa instancia. Sin embargo, en otros prompts sus puntuaciones fueron más moderadas, situándose entre 7 y 9, lo que sugiere una satisfacción algo inconsistente.

1. **CONCLUSIÓN**

La deuda técnica y la baja calidad del código son problemas críticos que afectan profundamente la eficiencia operativa y financiera de las empresas de software. A lo largo de este proyecto, se ha desarrollado un prototipo basado en inteligencia artificial, denominado agente JAN, enfocado en la detección y refactorización de código Java según los principios de Clean Code y las mejores prácticas de seguridad.

Los resultados obtenidos demuestran que la implementación de herramientas automatizadas como JAN puede reducir significativamente la deuda técnica. El agente no solo identifica de manera efectiva las violaciones a estándares de calidad y seguridad en el código, sino que también proporciona retroalimentación detallada y citas directas a normas relevantes, facilitando así la corrección por parte de los desarrolladores. Comparado con otros modelos existentes, JAN ofrece una precisión y profundidad de análisis superiores, lo que se traduce en ahorros sustanciales de tiempo y recursos.

La adopción de este prototipo permite optimizar los tiempos de desarrollo hasta en un 50% y mejorar la productividad en un 35%. Además, al fortalecer la seguridad del código, se mitigan riesgos asociados a vulnerabilidades críticas, como filtraciones de datos y ataques cibernéticos, protegiendo los activos de información y la reputación de la empresa.

En conclusión, la integración de inteligencia artificial en los procesos de desarrollo y

mantenimiento de software es una estrategia efectiva para abordar la deuda técnica y elevar la calidad del código. Este enfoque no solo reduce costos operativos y tiempos de entrega, sino que también mejora la seguridad y sostenibilidad de las soluciones tecnológicas. Futuras investigaciones podrían explorar la ampliación de este enfoque a otros lenguajes de programación y la incorporación de nuevas funcionalidades que aborden aspectos adicionales de la calidad del software.

**Bibliografía**

[1] Instituto Tecnológico de Costa Rica. (s.f.). *Análisis de casos de estudio*. RepositorioTEC. <https://repositoriotec.tec.ac.cr/handle/2238/9392>

[2] Universidad de Medellín. (s.f.). *Investigación sobre el desarrollo sostenible*. Repositorio UDEM. <https://repository.udem.edu.co/handle/11407/6256>

[3] Instituto Tecnológico de Buenos Aires. (s.f.). *Propuesta de optimización logística*. Repositorio ITBA. <https://ri.itba.edu.ar/server/api/core/bitstreams/e7dcb3f2-0e54-4074-9c87-56cc2b0f4db8/content>

[4] Universidad de Guadalajara. (s.f.). *Estudio de caso en administración pública*. Repositorio UdeG. <https://riudg.udg.mx/bitstream/20.500.12104/96392/1/MCUCEA11037FT.pdf>

[5] Universidad Nacional Autónoma de México. (s.f.). *Análisis de políticas públicas en México*. Repositorio UNAM. <https://ru.dgb.unam.mx/bitstream/20.500.14330/TES01000806473/3/0806473.pdf>

[6] Zhou, M., Zhang, L., & Zhu, H. (2019). A systematic review of empirical studies on technical debt. *IEEE Transactions on Software Engineering*, 45(5), 412-436.

[7] Spinellis, D., & Gousios, G. (2016). *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*. O'Reilly Media.

[8] Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the principal of an application's technical debt. In *Proceedings of the 3rd international workshop on managing technical debt* (pp. 1-6). ACM.

[9] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., ... & Sullivan, K. (2010). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 47-52). ACM.

[10] Pérez-Gutiérrez, J. C., & Correal, D. A. (s.f.). Un enfoque de Machine Learning para apoyar la mejora de la calidad del software mediante la refactorización automática de código. Semantic Scholar. Recuperado de <https://www.semanticscholar.org/paper/Un-Enfoque-de-Machine-Learning-para-Apoyar-la-de-la-P%C3%A9rez-Guti%C3%A9rrez-Correal/5b0884b0cbe38e9b56c4e5005d09bede556b66>

[11] Villar, M., & Matalonga, S. (s.f.). Definiciones y tendencia de deuda técnica: Un mapeo sistemático. Semantic Scholar. Recuperado de <https://www.semanticscholar.org/paper/Definiciones-y-Tendencia-de-Deuda-T%C3%A9cnica%3A-Un-Mapeo-Villar-Matalonga/1ea58f8d9dcbe35a15197cfcec919ac7889e05af>

[12] Hadad, S., & Sebastián, D. (s.f.). Inspecciones para mejorar la calidad de modelos en proyectos de software. Semantic Scholar. Recuperado de <https://www.semanticscholar.org/paper/Inspecciones-para-mejorar-la-calidad-de-modelos-en-Hadad-Sebasti%C3%A1n/63754e845b082c9a29b9d3faa9c1db42ee9d27f2>

[13] Digital55. (s.f.). Auditoría de código: Qué es y cuáles son sus beneficios. Recuperado de <https://digital55.com/blog/auditoria-de-codigo-que-es-y-cuales-son-sus-beneficios/>

[14] Informatec Digital. (s.f.). Auditoría de sistemas. Recuperado de <https://informatecdigital.com/sistemas/auditoria-de-sistemas/>

[15] Parasoft. (s.f.). Code quality solutions. Recuperado de <https://es.parasoft.com/solutions/code-quality/>

[16] Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., & Avgeriou, P. (2020). Can Clean New Code Reduce Technical Debt Density? Recuperado de <https://consensus.app/papers/code-reduce-technical-debt-density-digkas/984566ededb55cb59b6482606748e2f8/>

[17] Arif, A., & Rana, Z. (2020). Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort. Recuperado de <https://consensus.app/papers/refactoring-code-remove-technical-debt-reduce-arif/5738844d004b51f2becf4885dc381604/>

[18] Fuente propia Juan,M.., Andrés,M.., &Andrés, N.(2024). Pruebas realizadas a los agentes [Pruebas.xlsx](https://uniempresarial-my.sharepoint.com/:x:/g/personal/jmahechav_uniempresarial_edu_co/EYv2CRnikplJvvYFLlOzMpYBiDMBcruuJQ-wXMQqrRXrJA?e=N53ZaV)