



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

DEL DISEÑO AL CÓDIGO

LA PERSONA DETRÁS DEL SOFTWARE
EN LA ERA DE LA INTELIGENCIA ARTIFICIAL



ENTREVISTA • HOMENAJE

Álvaro Giovanni Longo Morales: trayectoria, vocación y legado



REMEMBRANZAS

ING. MARLON ORELLANA

Remembranza e historia con el Ing. Álvaro Longo

Todo inició con una persona a quien no había tenido la oportunidad de conocer y de quien sabía que recientemente había asumido el reto de dirigir la Revista de la Escuela de Ciencias y Sistemas.

Al recibir la invitación para integrar un equipo de trabajo que aportara algo diferente a la revista en conmemoración de los 50 años, me encontré con este personaje cuya calidez humana emanaba una paz y una tranquilidad sin igual.

Presentamos un proyecto más que ambicioso. Pusimos alma, mente y corazón en la Revista de los 50 años. Ese trabajo enlazó una amistad que trascendió a través de muchos años. Lo conocí como profesional y como un docente excepcional, consciente de que su misión de transmitir sus conocimientos dejaría marcada a muchas generaciones.

Con el tiempo interactuamos en diferentes ocasiones. Tuve la oportunidad de acercar mi familia a la de él y él la suya a la mía. No tengo palabras suficientes para decir: muchas gracias, Álvaro. Tu recuerdo permanece en nuestros corazones y tu ejemplo quedó grabado en muchas personas.

Hoy siento que hubiera querido compartir muchos años más, pero me quedo con todo lo bueno que él fue: una persona única, en momentos en los que él hace lo que él hizo; un sancharista excepcional, con una enorme calidad humana, una gran cercanía a Dios y un profundo compromiso con todo su entorno.

FLORIZA ÁVILA DE MEDINILLA

Profesora titular · Escuela de Ingeniería en Ciencias y Sistemas

Conocí al Ing. Álvaro Giovanni Longo Morales cuando retornó a la Facultad de Ingeniería con el propósito de desarrollar el proyecto con el que concluiría su proceso de graduación. En aquel momento, se requería el apoyo de estudiantes del Ejercicio Profesional Supervisado (EPS) para impulsar la elaboración de la revista de la Escuela de Ciencias y Sistemas. Sin dudarlo, él aceptó colaborar, iniciando una valiosa e inestimable contribución a dicho proyecto.

Lo recordaré siempre por su sonrisa amable y su intachable cortesía hacia cada uno de nosotros. Su partida deja en mí un entrañable recuerdo y una profunda gratitud por haber coincidido en la vida con una persona de su calidez humana.

Sin lugar a duda, su ausencia representa una irreparable pérdida para nuestra comunidad universitaria. Expreso mis más sinceras condolencias a la familia Longo Fuentes, anhelando que la unión y los valiosos recuerdos atesorados en sus corazones les brinden paz y consuelo.

Hasta que volvamos a encontrarnos, Ing. Álvaro Longo.

REMEMBRANZAS

INGA. JENNY MONTENEGRO

Coordinadora de la revista

Tuve la oportunidad de conocer al Ing. Longo primero como docente, y desde entonces lo recuerdo como un profesor excepcional. En sus clases no solo destacaba por sus conocimientos y la facilidad con la que los transmitía, sino también por su paciencia, amabilidad y disposición para orientar a sus estudiantes. Fue de esos docentes que dejan una huella que va más allá del aula.

Posteriormente, en 2021, tuve la oportunidad de acercarme nuevamente a él cuando aceptó que pudiera realizar mi Ejercicio Profesional Supervisado (EPS) en la Revista Ciencias y Sistemas. Durante esta etapa fue quien me guió y acompañó en cada paso, siempre con paciencia, brindándome consejos y compartiendo sus conocimientos. Más adelante, me hizo la invitación para formar parte del Comité Editorial de la revista, lo que me permitió conocerlo desde otra perspectiva y compartir más de cerca con él.

A través de estas experiencias pude conocer a una persona amable, inteligente y comprometida, que siempre buscaba el bien para los estudiantes y el crecimiento de nuestra Escuela. Para mí, fue una persona que confió en mi trabajo y me brindó oportunidades que marcaron de manera importante mi trayectoria.

Hoy, al recordar al Ing. Longo, me quedo con una profunda gratitud por haber coincidido con él en diferentes etapas de mi formación y por haber tenido la oportunidad de aprender no solo de sus conocimientos, sino también de su calidad humana. Su partida deja un vacío muy grande en nuestra Facultad y en quienes tuvimos la dicha de conocerlo. Sin embargo, creo que las personas que dejan huella nunca se van del todo: permanecen en cada enseñanza, en cada consejo y en cada oportunidad que brindaron. Yo llevaré siempre conmigo el recuerdo de aquel docente que me enseñó, del profesional que me guió y, sobre todo, de la persona que creyó en mí y me abrió las puertas para formar parte de algo que hoy tiene un significado aún más especial. Gracias, Ing. Longo, por haber sido parte de mi camino.

MIGUEL ANGEL ESTRADA CIFENTES

Ex estudiante

Un ingeniero que se preocupaba por enseñar y dar lo mejor a sus estudiantes, atento al aprendizaje de las siguientes generaciones y una persona caracterizada por su humildad.

PABLO DANIEL FERNÁNDEZ CHACÓN

Editor y Epesista de la revista

Como parte de mi formación en esta carrera tuve el privilegio de recibir clases con el ingeniero en Bases de Datos 1; se notó siempre su entrega y, aun en el cansancio, daba lo mejor de sí para ofrecer una buena clase. Recuerdo muy bien sus consejos y, sobre todo, una frase que marcó a muchos de nosotros:

"Aquí se gradúa el necio, el persistente, aquel que a pesar de ir mal da lo mejor de sí mismo; he tenido muchos compañeros que se fueron, que no se graduaron, que eran mucho más inteligentes que yo y aun así no llegaron a ser ingenieros. Yo, que fui repechón, soy ingeniero; no se rindan muchá, ahorita están en la primer cima, tomen un poquito de aire, porque ahorita es donde más deben, en vez de ir más despacio, ir constante y tal vez acelerar un poquito más para terminar al fin: yo sé que ustedes pueden".

Nunca tuve la oportunidad de agradecerle tales palabras y hoy se nos fue, dejando no solo un espacio muy difícil de llenar en la Escuela, sino también un legado imborrable en innumerables generaciones por su entrega y su forma única de ser. Hasta la próxima, inge; esperamos que Dios nos permita volvernos a ver.

BRIAN ALEXANDER GARCÍA ORR

Ex estudiante

Quien dejó una huella imborrable en nuestras vidas. Su entusiasmo incansable, su forma tan amena de transmitir conocimientos y su constante esfuerzo para que cada uno de nosotros saliera adelante son un legado invaluable. Gracias por impulsarnos a superar cada dificultad con sus palabras de ánimo. Su vocación vivirá siempre en sus estudiantes. Descanse en paz.

AUTORIDADES Y EQUIPO EDITORIAL

Autoridades

Escuela de Ingeniería en Ciencias y Sistemas



Comité editorial

DIRECTOR DE ESCUELA

Msc. Ing. Carlos Alonzo

COORDINACIÓN EDITORIAL · PM DE REVISTA

Msc. Inga. Jenny Montenegro

Dirección editorial, coordinación y gestión de la edición

COLABORACIÓN ESPECIAL

Ing. Miguel Marin de León

DISEÑO · EDICIÓN · DIAGRAMACIÓN

Pablo Daniel Fernández Chacón

EDITORIAL

Después del asombro, la responsabilidad

Edición XXIX · Del diseño al código

Ya no se trata de decidir si usaremos inteligencia artificial, sino de entender qué valor aportamos cuando la herramienta puede hacer parte del trabajo.

La inteligencia artificial dejó de ser una promesa lejana. Hoy redacta, resume, programa, clasifica, diseña, recomienda, traduce y analiza a una velocidad que hace pocos años habría parecido improbable. Está en las aulas, en los proyectos de los estudiantes y en la práctica diaria de los docentes. Se utiliza porque funciona: acelera tareas y, bien empleada, amplía lo que una persona o un equipo puede lograr.

Una escuela de ingeniería no puede quedarse en el asombro; su tarea comienza después. Cuando una herramienta genera código, propone arquitecturas, sintetiza documentos o explora millones de registros en segundos, es necesario preguntarse con honestidad qué significa hoy formar a un ingeniero. Su valor ya no puede medirse solo por la velocidad con la que ejecuta una tarea.

El papel del profesional se desplaza hacia lo que la herramienta no puede asumir por él: plantear el problema correcto, entender el contexto, definir restricciones, distinguir una respuesta plausible de una respuesta válida, proteger a las personas afectadas por cada decisión y responder por las consecuencias de lo que construye. La inteligencia artificial puede proponer, pero el criterio sigue siendo humano. Puede optimizar, pero alguien debe decidir qué merece optimizarse. Puede generar una solución, pero alguien debe juzgar si esa solución debía existir, y si debía existir de esa manera.

Ese criterio no se descarga ni se automatiza: se forma. Y formarlo es, hoy más que nunca, la razón de ser de la escuela.

Por eso, esta Edición XXIX no presenta a la inteligencia artificial como un genio omnisciente ni como una amenaza inevitable, sino como lo que es: una herramienta extraordinariamente poderosa. Y toda herramienta poderosa exige profesionales más conscientes, no menos. Los datos requieren gobernanza; los requerimientos, comprensión; la arquitectura, propósito; la seguridad, responsabilidad; el código, criterio; la documentación, continuidad; y los equipos, liderazgo.

Las entrevistas y los artículos que integran estas páginas recorren ese cambio desde distintos ámbitos del oficio. No se limitan a preguntar qué puede hacer la inteligencia artificial: indagan qué corresponde hacer ahora que está aquí. Esa es, en última instancia, la pregunta que define esta edición y que desafía a toda la comunidad académica: no cuánto puede hacer una máquina por nosotros, sino qué clase de ingenieros decidimos ser, y formar, cuando ya no tenemos que hacerlo todo con nuestras propias manos.

Msc. Ing. Carlos Alonzo

Director de la Escuela de Ciencias y Sistemas
Universidad de San Carlos de Guatemala

Índice general

Homenaje · Álvaro Giovanni Longo Morales	2-3
Autoridades y equipo editorial	4
Editorial - Después del asombro viene la responsabilidad	5
SECCIÓN 1 · Gobernanza de datos y responsabilidad humana	7
Entrevista · Kevin Aragón	8
Cuando la IA se equivoca: ¿quién responde?	10
El dilema de los privilegios	12
SECCIÓN 2 · Análisis y arquitectura	14
Entrevista · Julio Marroquín	15
Programar no es suficiente	17
De la visión al código	19
La arquitectura que nadie pidió	21
Reaccionar a tiempo	23
Arquitecto de la torre de marfil	25
Comunicación efectiva entre analistas, arquitectos y programadores	27
SECCIÓN 3 · Seguridad, despliegue y operación	29
Entrevista · Marlon Francisco Orellana López	30
La seguridad no se agrega al final	32
Cómo evitar que la responsabilidad compartida se diluya	34
Compila hoy, rompe mañana	36
SECCIÓN 4 · Programación e inteligencia computacional	38
Entrevista · Luis Fernando Cajas	39
¿Reemplazo o transformación? El impacto de la IA en la programación	41
Generado por IA, aprobado por ti	43
El fin del desarrollo de software tradicional	45
SECCIÓN 5 · Personas, documentación y continuidad	47
Entrevista · Víctor Castellanos	48
Del desarrollador al equipo: conocimiento para un software que perdure	50



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

SECCIÓN 1

Gobernanza de datos y responsabilidad humana

¿Quién decide cómo se obtienen, utilizan, protegen y conservan los datos?

ARTÍCULOS DESTACADOS



Profesional de datos antes y después de la IA



Calidad, procedencia y trazabilidad



Datos, memoria y permisos de agentes



Privacidad, consentimiento y acceso



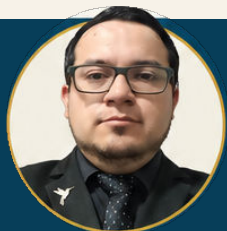
Responsabilidad ante decisiones automatizadas



ENTREVISTA · PROFESIONAL

Kevin Aragón

Analítica y gobierno de datos



TEMA

**Gobernanza de datos y
responsabilidad humana**


Ver la entrevista

PRESENTACIÓN

¿Quién es Kevin Aragón y qué le atrae del trabajo con datos?

Kevin Aragón acumula más de una década entre analítica y gobierno de datos. Le atrae que ningún proyecto sea idéntico al anterior: cada organización tiene problemas, fuentes y decisiones distintas. Para él, trabajar con datos no consiste en repetir una receta, sino en razonar, descubrir hallazgos y entender el negocio lo suficiente como para convertir información dispersa en algo útil para quien debe decidir.

ANTES

Antes de incorporar inteligencia artificial, ¿cómo se definía qué información recopilar y cuánto tiempo conservarla?

El trabajo era mucho más manual y estaba limitado por la capacidad de almacenamiento disponible. Había que identificar qué historia resultaba relevante, reunir información desde varias fuentes y decidir qué debía conservarse. En analítica, Kevin señala que con frecuencia cinco o seis años ofrecen una ventana útil, aunque algunos clientes pidan más. Después venía la integración: unir datos de distintas fuentes para llevarlos a un modelo de data warehouse, estrella, copo de nieve o una estructura híbrida.

ANTES

¿Cuáles eran los principales problemas de calidad, procedencia y trazabilidad?

El problema nunca fue solamente tener pocos datos; también puede existir un enorme volumen con muy poca calidad. La primera tarea es reconocer cuáles fuentes considera confiables el cliente y cuáles son complementarias. A partir de ahí entran limpieza, transformación, reglas y gobierno de datos hasta construir una versión coherente del registro. Esa búsqueda de un Golden Record permite que la siguiente etapa -el análisis- parta de información estructurada y no de una colección de valores contradictorios.

TRANSICIÓN

¿Qué cambió primero cuando la IA comenzó a entrar en la gestión de datos?

Antes, buena parte del ETL exigía conversar con el cliente, entender su problemática y construir manualmente las transformaciones. Hoy una herramienta puede recibir contexto y proponer código, consultas o análisis en segundos. Para Kevin, la diferencia fundamental está en la velocidad, no en la responsabilidad: alguien debe revisar si la transformación es correcta, si la respuesta corresponde al problema y si la herramienta entendió realmente la estructura del negocio.

TRANSICIÓN

¿Cómo determina si un conjunto de datos es suficientemente confiable antes de utilizarlo?

La validación sigue descansando en métricas, reglas y conciliación. Si se pide analizar cinco años de ventas, no basta con aceptar el resultado porque esté bien presentado: hay que comprobar de qué tablas salió, si los totales cuadran, si las reglas de negocio se respetan y si la fuente utilizada era la correcta. La IA puede entregar una respuesta convincente aun cuando haya entendido mal la petición; por eso el profesional debe razonar sobre el resultado y no limitarse a admirar su forma.

ENTREVISTA • CONTINUACIÓN

Kevin Aragón

Continuación

DESPUÉS

¿Qué debería hacerse cuando los datos están incompletos, desactualizados o presentan sesgos?

Kevin propone enriquecerlos cuando sea posible. Si una dirección solo incluye departamento y municipio, pueden buscarse dimensiones u otras fuentes estructuradas que permitan completar relaciones útiles. Cuando el dato no existe o no puede recuperarse, debe quedar identificado como no asignado en lugar de inventarse. La prioridad es que la información conserve consistencia y trazabilidad. Completar un vacío con una suposición puede hacer que un reporte se vea completo, pero deja de ser confiable.

DESPUÉS

¿Cómo decidir qué información puede consultar, recordar, modificar o compartir un agente?

No todos los agentes operan bajo las mismas condiciones. Un agente local y uno conectado a un servicio público implican riesgos distintos. Hay que considerar políticas internas, legislación, ubicación de los datos y acuerdos de seguridad. Kevin utiliza el ejemplo del sector bancario: cierta información no puede abandonar determinadas jurisdicciones. En esos escenarios entran cifrado, controles de acceso y delimitación clara de qué puede consultar la herramienta y qué debe permanecer aislado.

DESPUÉS

¿Cómo deberían manejarse privacidad, consentimiento y mínimo privilegio?

La costumbre de aceptar permisos sin leerlos se vuelve especialmente peligrosa cuando una herramienta puede inferir mucho más de lo que el usuario imagina. Ubicación, rutinas y datos aparentemente pequeños pueden revelar patrones personales. A nivel empresarial sucede algo parecido: cada arquitecto, desarrollador o técnico debe responder por la información que manipula. El principio de mínimo privilegio obliga a entregar únicamente el acceso necesario para una tarea concreta y durante el tiempo necesario.

RESPONSABILIDAD

¿Qué acciones deberían conservar autorización humana obligatoria?

Kevin considera que la mayoría de decisiones relevantes necesitan supervisión. Un agente puede asumir tareas de analista, arquitecto o técnico, pero solo responde al conocimiento y contexto que recibió. No conoce por sí solo las convenciones del cliente, sus restricciones, formatos o prioridades. Puede producir un resultado técnicamente correcto y aun así ajeno a la necesidad real. Mantener una persona revisando no significa rechazar la automatización: significa colocarla dentro de un proceso responsable.

CIERRE

Si una decisión automatizada resulta incorrecta, ¿quién responde?

La responsabilidad no desaparece detrás de la herramienta. Los datos organizacionales siguen perteneciendo a la organización y los datos personales requieren protección especial. Si un proyecto toma una mala decisión por una fuente deficiente o una automatización mal revisada, el equipo que gestionó y aprobó ese proceso debe responder. Para Kevin, la IA ayuda, pero no suplanta a quienes diseñan el sistema. Precisamente por eso la gobernanza sigue siendo una tarea humana.

IDEA CENTRAL

La IA puede acelerar el análisis; la confianza sigue dependiendo de datos gobernados y de una persona capaz de validar lo que recibe.

Cuando la IA se equivoca: ¿quién responde?

Autores: José Estuardo Orellana Leonardo; Emiliano José Velásquez Najera

Palabras clave: inteligencia artificial, decisiones automatizadas, responsabilidad, supervisión humana, trazabilidad.

Introducción

Planteemos el siguiente escenario: Una persona solicita un crédito, su solicitud es rechazada en unos segundos. Ningún humano revisó su expediente, sino un sistema realizó el análisis de sus datos y dio un resultado. En primera instancia, parece una decisión objetiva tomada por un sistema automatizado, no obstante, surge una pregunta: ¿Qué pasa si la decisión es incorrecta?

En la actualidad, la inteligencia artificial ya genera recomendaciones, detecta fraudes, clasifica información y realiza otros procesos cotidianos. Sin embargo, cuando esta automatización comienza a intervenir en decisiones con consecuencias reales para personas reales, surge un problema. No basta únicamente con preguntar si el sistema funciona, es necesario identificar al diseñador del sistema, a su supervisor y quién es el responsable cuando existen fallas.

Desarrollo

Las decisiones no son completamente automáticas. Para que un modelo produzca resultados, se tuvo que haber determinado el problema, recolectar los datos, establecer reglas y definir el curso de las acciones a ejecutar por el sistema. Debido a esto, la responsabilidad no se desvanece cuando se ejecuta por una inteligencia artificial, únicamente cambia el lugar dentro del proceso.

Este problema puede surgir en cualquier etapa del proceso. Los datos pueden estar incompletos, las reglas de negocio pueden haber sido mal definidas o alguna configuración específica pudo permitir al sistema tomar una decisión que no debía automatizarse.

En la actualidad, esto ya ocurre en sistemas utilizados para recomendaciones, detección de fraude y análisis de comportamiento. En un entorno de producción, pueden aparecer situaciones que no fueron previstas en la etapa de pruebas, en especial cuando los datos reales difieren a los que fueron utilizados en la fase de entrenamiento.

¿Quién tiene la responsabilidad?

No es “si es que”, sino “cuando” la IA se equivoca, una explicación simple puede ser responsabilizar al programador, o en su defecto, al modelo. No obstante, la responsabilidad debe distribuirse entre varias personas y áreas. En retrospectiva, el analista pudo haber definido incorrectamente un requerimiento, o un desarrollador pudo haber implementado una lógica defectuosa, o el administrador pudo haber configurado mal el sistema, incluso el usuario pudo haber aceptado

una recomendación sin verificarla, también puede ser el caso que la organización haya puesto en producción una solución sin el control suficiente.

Existe otra dificultad cuando no es posible reconstruir o rastrear como se obtuvo un resultado. Por ejemplo, cuando un sistema rechaza una transacción, debería quedar registro de los datos que utilizó, la versión del modelo que lo ejecutó y la configuración de ese momento.

En este sentido, la trazabilidad es fundamental, los registros de los eventos, las versiones del modelo, si existen cambios de configuración y las bitácoras hacen posible reconstruir lo que ocurrió después de un incidente. De tal forma que no son únicamente herramientas para solucionar problemas técnicos, también permiten establecer responsabilidad.

Supervisar antes de lamentar

Supervisar una IA no implica revisar manualmente cada resultado, sino definir qué decisiones pueden automatizarse por completo y cuales aún requieren intervención humana.

El marco de gestión de riesgos de inteligencia artificial de NIST (National Institute of Standards and Technology) propone cuatro funciones: Govern, Map, Measure y Manage, que permiten identificar riesgos durante el ciclo de vida del sistema. Esto resulta útil porque convierte la supervisión en una actividad continua y no en un elemento reactivo posterior a una falla.

En Ingeniería en Ciencias y Sistemas, la supervisión debe considerarse desde la etapa de diseño. El sistema puede requerir alertas, límites, permisos o mecanismos para detener una operación al identificarse una condición de riesgo.

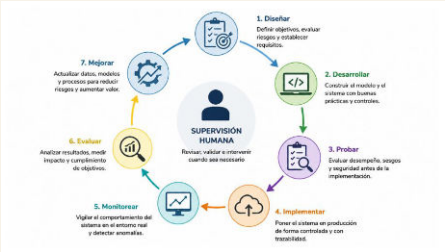


Figura 1. Supervisión humana durante el ciclo de vida de la IA. En cada etapa debe existir un responsable, junto con mecanismos para revisar resultados, registrar cambios e intervenir cuando sea necesario.

Fuente: elaboración propia a partir del NIST AI Risk Management Framework.

No se hace un llamado a detener la automatización, por el contrario, automatizar puede mejorar la eficiencia y eliminar la

CONTINUACIÓN · ARTÍCULO

Cuando la IA se equivoca: ¿quién responde?

realización manual de tareas repetitivas. Sin embargo, se debe evitar que la automatización se vuelva una excusa y no se definan responsabilidades. La OCDE (Organización para la Cooperación y el Desarrollo Económicos) en sus principios de inteligencia artificial incorpora la transparencia, seguridad y rendición de cuentas como elementos necesarios para desarrollar sistemas confiables¹. Así mismo, la regulación europea ha establecido requisitos de supervisión humana para determinados sistemas considerados de alto riesgo o no delegables².

Lo anterior expuesto muestra que la responsabilidad no debería aparecer únicamente cuando los sistemas fallan, sino que debe formar parte de la arquitectura, los procesos e incluso la documentación desde el principio de la implementación del sistema.

Conclusiones

La inteligencia artificial puede producir decisiones, pero no se puede hacer responsable por las consecuencias de esas decisiones. Detrás de cada sistema hay datos, reglas, código, configuraciones y lo más importante, personas, que intervienen en distintas etapas. Por lo anterior, ante un fallo, no puede delegarse un único responsable, es necesaria una trazabilidad para reconstruir el resultado que llevó a una decisión y determinar la etapa que falló. Es necesaria la supervisión humana desde el principio y cuantificar el riesgo de cada sistema.

Cuando se automatiza una tarea, no se elimina el criterio profesional, este se traslada hacia el diseño, validación y control. El nuevo reto para el diseñador del sistema no es conseguir que el modelo funcione, sino construir sistemas que puedan ser comprendidos, rastreables y corregidos. No se aconseja ir contra la corriente, puesto que la tecnología seguirá avanzando y los procesos de decisiones automatizadas aumentarán en frecuencia. No obstante, estas decisiones continuarán afectando personas y organizaciones, así que deberá existir alguien que pueda responder por ellas.

Referencias

Organisation for Economic Co-operation and Development. "AI Principles." OECD. Consultado en agosto de 2026. <https://www.oecd.org/en/topics/ai-principles.html>.

Tabassi, Elham. Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 1001. Gaithersburg, MD: National Institute of Standards and Technology, 2023. <https://doi.org/10.6028/NIST.AI.100-1>.

UNESCO. Recomendación sobre la Ética de la Inteligencia Artificial. París: UNESCO, 2022. <https://www.unesco.org/es/artificial-intelligence/recommendation-ethics>.

Unión Europea. Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act). EUR-Lex,

2024. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:32024R1689>.

1 OCDE: Organisation for Economic Co-operation and Development, "AI Principles," OECD, consultado en agosto de 2026, <https://www.oecd.org/en/topics/ai-principles.html>.

2 NIST: Elham Tabassi, Artificial Intelligence Risk Management Framework (AI RMF 1.0), NIST AI 100-1 (Gaithersburg, MD: National Institute of Standards and Technology, 2023)

El dilema de los privilegios: antes y después de los agentes autónomos de IA

Autores: Marvin Geobani Pretzantzín Rosalío; Omar Alejandro Vides Esteban

Palabras clave: Agente, seguridad, identidad, gobernanza

Introducción

La inteligencia artificial ha demostrado un potencial transformador en múltiples sectores, impulsada en gran medida por el uso de agentes autónomos: software diseñado para ejecutar directivas complejas utilizando un modelo de lenguaje como centro de mando. Sin embargo, para operar, estos sistemas requieren permisos de lectura, escritura y ejecución.

La ciberseguridad tradicional asume usuarios humanos. Con la llegada de los agentes, ocurre un cambio de paradigma: gestionar entidades no humanas que operan a alta velocidad, sin fatiga y privilegios por defecto excesivos. Este artículo analiza cómo este dilema de privilegios altera el rol del especialista en seguridad y expone los riesgos de una automatización desatendida.

¿Por qué cambia la seguridad con agentes?

Los incidentes de seguridad relacionados con la IA son cada vez más frecuentes debido a la falta de preparación organizativa ante el cambio tecnológico. Como muestran datos recientes de la industria (Proofpoint, Inc. 2026), la adopción tecnológica supera con creces la cobertura de control (ver Figura 1).

cadena de delegación entre sub-agentes o el uso de credenciales persistentes en la nube, agrandan la superficie de ataque al propagar tokens de acceso sin aislamiento de contextos (Sotiropoulos, Katz, y Del Rosario 2026).

El OWASP Agentic/ASI Top 10 (2026) identifica el abuso de identidad y privilegio del agente como un vector crítico. A diferencia de un usuario tradicional, un agente comprometido no requiere vulnerar llaves criptográficas; le basta con manipular la lógica de decisión del sistema (Sotiropoulos, Katz, y Del Rosario 2026).

Esta deficiencia de gobernanza propicia dos fallas secundarias el envenenamiento de memoria, donde una base de conocimientos RAG contaminada hace que el agente asimile instrucciones hostiles de manera puramente semántica, evadiendo escáneres estáticos; y las fallas en cascada, cuando tokens OAuth sin segmentación permite que un sub-agente escale privilegios hacia el nodo raíz (Sotiropoulos, Katz, y Del Rosario 2026).

Credenciales persistentes en entornos cloud

Para comprender el impacto de este dilema podemos representar una posible vulnerabilidad. El recurso: Claves de API y tokens de identidad (IAM) de larga duración se asignan a agentes de backend desatendidos, los cuales cuentan con permiso de lectura y escritura sobre bases de datos de producción.

El vector de ataque conocido es la Exposición de Identidad y Credenciales de Agente

(DSGAI02; OWASP GenAI Data Security Initiative 2026) donde un atacante inyecta instrucciones maliciosas en el flujo de entrada para forzar al agente a divulgar datos sensibles (CWE-732). Esto brinda al atacante la capacidad de ejecutar comandos maliciosos durante un periodo de tiempo indefinido.

Para poder evitar este tipo de casos es necesario implementar políticas de privilegio mínimo, rotación de tokens de corta duración y aislamiento de contextos de ejecución.



Figura 1. Preparación organizacional en seguridad de IA frente a su nivel de adopción. Fuente: Proofpoint, Inc. 2026.

Patrones comunes del error

Al gestionar identidades digitales, un error recurrente es el uso de entidades genéricas de alto privilegio para tareas generales, al fallar en delimitar las tareas del agente y asignarle más poder del necesario (Wilson y Lambros 2026). Otros patrones, como la confianza ciega en

Independencia y control: El rol del “Human-in-the-Loop”

El problema fundamental radica en que los sistemas de IA actúan con total independencia sin puntos de verificación humana que validen la intencionalidad de sus comandos. Cuando un agente posee la facultad de modificar controles de acceso sin supervisión, puede crear puertas traseras legítimas ante los ojos del sistema operativo (CWE-285; OWASP GenAI Data

Security Initiative 2026). Por esto es necesario establecer puntos de control donde una persona encargada realice validaciones de las actividades de los agentes y corrija el curso, evitando así conductas indeseables por parte de los agentes.

Conclusiones

El paso de gestionar entidades humanas a entidades autónomas exige una transformación radical en la ciberseguridad moderna. La automatización desprovista de una matriz estricta de

privilegios convierte al propio asistente inteligente en el eslabón más vulnerable de la organización (Proofpoint, Inc. 2026). El trabajo del especialista en seguridad ya no consiste únicamente en proteger puertas de acceso tradicionales, sino en limitar, auditar y delimitar de

forma rigurosa la autonomía de los agentes.

Referencias

OWASP GenAI Data Security Initiative. 2026. "OWASP GenAI Data Security: Risks and

Mitigations 2026." OWASP GenAI Security Project, v1.0, marzo 2026.

<https://genai.owasp.org/resource/owasp-genai-data-security-risks-mitigations-2026/>.

Proofpoint, Inc. 2026. "Proofpoint Research Reveals Half of Global Organizations

Experienced AI Incidents Despite Having AI Security Controls in Place." Proofpoint US,

abril 28, 2026.

<https://www.proofpoint.com/us/resources/threat-reports/ai-human-risk-landscape-report>.

Sotiropoulos, John, Keren Katz, y Ron F. Del Rosario. 2025. "OWASP Top 10 for Agentic

Applications 2026." OWASP GenAI Security Project, v1.0, diciembre 2025.

<https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/>.

Wilson, Sandy, y Ryan Lambros. 2026. "OWASP Top 10 for LLM Applications 2026."

OWASP GenAI Security Project, v1.0. <https://genai.owasp.org/>.



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

SECCIÓN 2

Análisis y arquitectura

¿Cómo se convierte una necesidad humana en una solución técnica coherente?

ARTÍCULOS DESTACADOS



El analista de software antes y después de la IA



De las necesidades humanas a los requerimientos técnicos



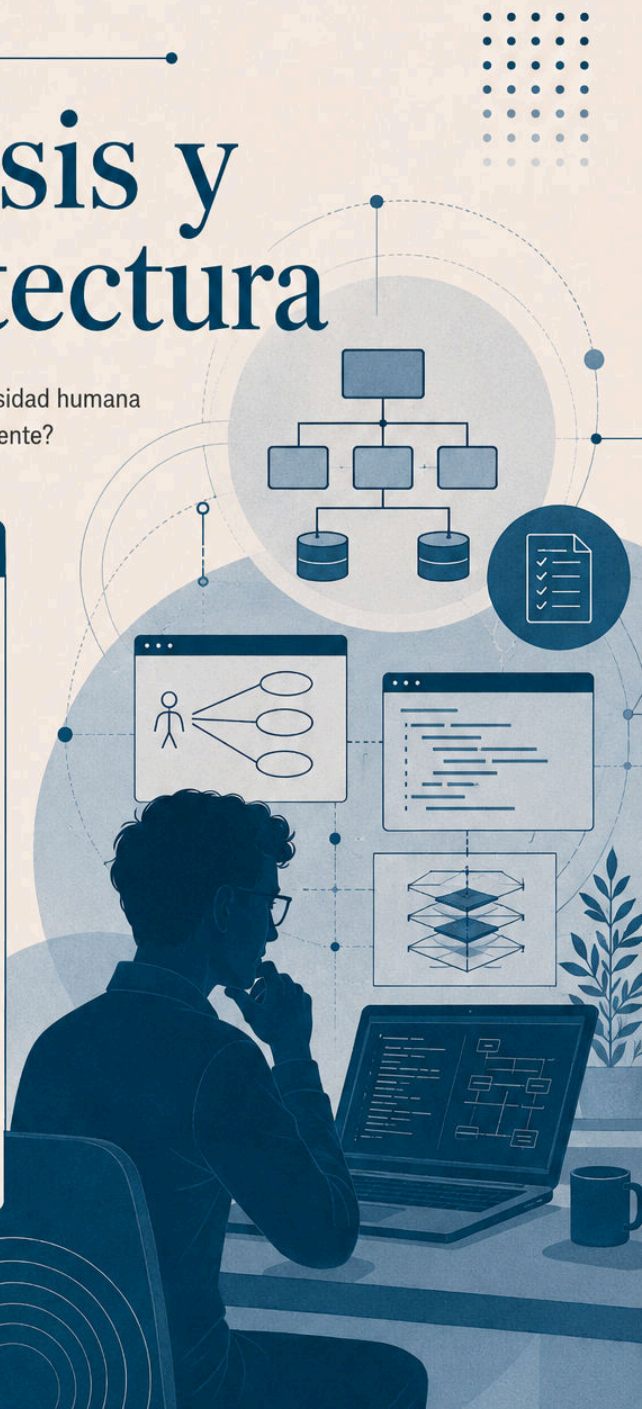
Del análisis funcional a la arquitectura



Comunicación entre analistas, arquitectos y programadores



IA para analizar requerimientos: apoyo y supervisión



ENTREVISTA • PROFESIONAL

Ing. Julio Marroquín

Analista de sistemas



TEMA

Del requerimiento a la arquitectura



Ver la entrevista

PRESENTACIÓN

¿Qué papel ocupa el analista entre el usuario, la arquitectura y el desarrollo?

Julio Marroquín lleva alrededor de diez o doce años trabajando en análisis de sistemas y describe el rol como un puente que muchas veces se subestima. El analista es la cara frente al usuario y, al mismo tiempo, la voz que lleva ese contexto hacia arquitectos y programadores. Su trabajo consiste en traducir lo que la persona solicita, lo que el proceso realmente necesita y lo que el equipo técnico puede convertir en una solución sostenible.

ANTES

Antes de utilizar asistentes de IA, ¿cómo realizaba el levantamiento y la validación de requerimientos?

El punto de partida eran reuniones extensas con usuarios y responsables del proceso. Se tomaban notas a mano, se solicitaba documentación y después se elaboraban diagramas que mostraban qué hacía el usuario, qué debía hacer el sistema y cuál sería el resultado. Era un trabajo completamente manual: hojas, lapiceros, dibujos y explicaciones sencillas para confirmar que el equipo había entendido lo mismo que la persona solicitante.

ANÁLISIS

¿Qué diferencia establece entre preanálisis y análisis?

En su forma de trabajo, el preanálisis consiste en comprender el proceso y producir un primer entregable que permita conversar: un prototipo. El análisis comienza cuando ese prototipo se une con las reglas de negocio, restricciones y giro de la organización. Esa distinción evita saltar demasiado rápido a la solución técnica. Primero se hace visible lo que creemos haber entendido; después se comprueba si realmente puede convertirse en un sistema coherente.

ANTES

¿Qué parte consume más tiempo y qué habilidades son indispensables?

Entender procesos sigue siendo una de las tareas más costosas. Muchas personas desempeñan un puesto sin tener documentado el proceso completo, por lo que el analista debe reconstruirlo mediante conversación. Allí aparecen las habilidades blandas: escuchar, preguntar, ordenar y, sobre todo, encauzar la reunión. Una sesión puede convertirse fácilmente en un desahogo sobre el sistema actual; el analista debe reconocer esa frustración sin perder el objetivo de la toma de requerimientos.

ENTREVISTA • CONTINUACIÓN

Ing. Julio Marroquín

Continuación

TRANSICIÓN

¿Cuál fue la primera actividad que cambió con la llegada de la IA?

El prototipado. Antes se trabajaba con wireframes o mockups de baja fidelidad: dibujos planos, sin colores y casi sin experiencia de usuario. Con herramientas generativas, esos mismos bocetos comenzaron a convertirse rápidamente en prototipos funcionales en HTML. Para Julio el cambio fue importante porque el usuario dejó de imaginar el comportamiento y pudo abrir el prototipo en un navegador, navegarlo y reconocerse dentro de la solución. La validación pasó de describir a experimentar.

TRANSICIÓN

Quando el cliente expresa lo que quiere, ¿cómo separa la solicitud literal de la necesidad real?

La experiencia le enseñó a volver siempre al proceso. Un usuario puede pedir reportes, biometría o funciones espectaculares porque las vio en otra aplicación, pero el sistema debe responder primero a lo que su operación necesita. Julio resume la idea con una regla práctica: automatizar procesos, no costumbres. Una práctica repetida durante años no necesariamente es un proceso bien definido ni una norma que el nuevo sistema deba conservar.

DESPUÉS

¿Qué papel tiene la documentación en esa transición entre usuario y equipo técnico?

Aunque reconoce que la documentación suele ser un punto incómodo para muchos equipos, también aparece como una condición para comprender el proceso. Pedir al usuario documentos, representar flujos y dejar una explicación de lo entendido permite que la conversación no dependa solo de la memoria. El documento no sustituye el diálogo; lo vuelve verificable. Sirve para comparar lo solicitado con lo que arquitectos y programadores terminarán construyendo.

DESPUÉS

¿Cómo influye el presupuesto en la arquitectura que puede proponerse?

La factibilidad forma parte del análisis desde temprano. Si alguien pide recuperar una contraseña mediante lectura de retina, técnicamente puede estudiarse, pero la conversación cambia cuando se muestran el costo de implementación, mantenimiento, especialistas e infraestructura. El presupuesto obliga a priorizar. En lugar de intentar construir una versión gigantesca desde el primer día, el analista ayuda a definir qué se necesita ahora y qué puede quedar para una versión futura.

CIERRE

¿Qué permanece esencial aunque la IA pueda generar prototipos y documentación?

El criterio para comprender el proceso y la capacidad de conversar con personas. Una herramienta puede dibujar, resumir o producir HTML, pero no reemplaza la responsabilidad de descubrir qué problema vale la pena resolver. El analista sigue siendo quien mantiene alineadas intención, reglas de negocio, restricciones y expectativas. La IA acelera el artefacto; el trabajo profesional consiste en conservar el significado que ese artefacto debe representar.

IDEA CENTRAL

La IA puede acelerar el prototipo. El analista sigue siendo responsable de comprender el proceso que ese prototipo representa.

ARTÍCULO · ESTUDIANTE

Programar no es suficiente: comprender el problema antes de diseñar el sistema

Autores: Carlos Manuel Lima y Lima; Josseline Griselda Montecinos Hernández

Palabras clave: Requerimientos, usuarios finales, dominio del negocio, historias de usuario.

Introducción

Hoy en día, desarrollar software es más rápido que antes. Las herramientas de inteligencia artificial pueden generar código, proponer soluciones e incluso ayudar a construir sistemas en

poco tiempo. Sin embargo, que un sistema funcione técnicamente no significa que resuelva el problema para el que fue creado. Antes de programar, el ingeniero necesita comprender el negocio, los usuarios, sus reglas y necesidades.

De aquí surge una pregunta clave: si el ingeniero puede programar una solución pero no comprende el problema, ¿realmente está desarrollando el sistema correcto? Para evitar esto,

conocer el dominio del negocio y transformar las necesidades del usuario en requerimientos adecuados es fundamental. En definitiva, el desarrollo no depende únicamente de escribir código, sino de entender correctamente el problema desde el principio.

Desarrollo

Cuando un usuario necesita que los ingenieros resuelvan un problema, él ya tiene definido cuál es su problema, porque conoce su negocio, ha interactuado con sus clientes y sabe todas sus reglas y restricciones. Cuando transmite sus necesidades, la mayoría de ingenieros no son expertos en el negocio; el experto es el solicitante. Pero no basta solo con escucharlo, para dar una buena solución los ingenieros también deben conocer y comprender el negocio. La toma de requerimientos no debe ser platicar con el cliente y escuchar que quiere un sistema para monitorear sus ganancias. Los ingenieros deben comprender el área en la que trabaja, conocer cuáles son sus ingresos y egresos, de qué forma obtiene sus materias primas, cómo las vende al cliente, entre otras cosas. Solo al comprender estos procesos es posible identificar qué información necesita realmente el sistema y cuáles son los requerimientos que deben considerarse durante su desarrollo. El conocimiento del dominio no es un aspecto secundario durante la toma de requerimientos. El

International Institute of Business Analysis establece que, al preparar una actividad de toma de

requerimientos, se debe considerar el dominio del negocio y la participación de expertos en el área, ya que estos pueden aportar información y orientar sobre las fuentes que deben consultarse.¹ Esto permite comprender que el trabajo del ingeniero no consiste únicamente en recopilar lo que el usuario expresa, sino en analizar la información obtenida y transformarla posteriormente en requerimientos y diseños.²

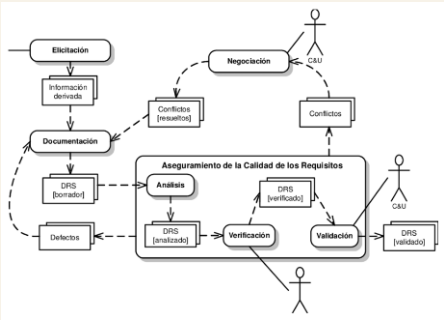


Ilustración 1. Modelo simplificado del proceso de ingeniería de requisitos. Fuente: Bernárdez, Beatriz; Amador Durán Toro y Miguel Toro (2003), consultado en ResearchGate.

ResearchGate.

Por esta razón, el conocimiento del dominio adquiere importancia durante la toma de requerimientos, ya que permite al ingeniero comprender las reglas, procesos y restricciones que

existen dentro de un determinado entorno. No se trata únicamente de saber qué sistema quiere el cliente, sino de comprender por qué lo necesita y cómo funciona el contexto en el que será utilizado. A partir de esta relación entre el conocimiento del dominio y la ingeniería de requisitos, se puede analizar hasta qué punto un ingeniero debe involucrarse en el área para la que desarrolla una solución.

Un requisito puede parecer claro para el usuario, pero resultar ambiguo para quien debe convertirlo en una solución técnica. Por ejemplo, solicitar “un sistema para monitorear las ganancias” no permite determinar qué

CONTINUACIÓN · ARTÍCULO

Programar no es suficiente: comprender el problema ante

información debe considerar el sistema ni cómo debe

realizarse el cálculo. Después de analizar el negocio, este requisito podría precisar los ingresos,

egresos, costos y ventas que deben registrarse y relacionarse para obtener la información solicitada. Este análisis permite transformar una necesidad general en un requerimiento más preciso y útil para las siguientes etapas del desarrollo.³

La inteligencia artificial también apoya durante la ingeniería de requerimientos, para organizar información, analizar historias de usuario, identificar posibles inconsistencias y proponer elementos relacionados con el dominio. Sin embargo, estas herramientas no sustituyen la participación del ingeniero, quien debe revisar los resultados y comprobar que la información obtenida corresponde con las necesidades, reglas y restricciones del negocio.³

Conclusiones

Comprender el problema antes de desarrollar permite que el sistema responda mejor a lo que realmente necesita el usuario. No se trata únicamente de recopilar solicitudes, sino de entender

cómo funciona el negocio, qué restricciones existen y qué situaciones deben tomarse en cuenta para convertirlas en requerimientos útiles. El ingeniero no necesita convertirse en experto del negocio, pero debe comprenderlo lo suficiente para traducir sus necesidades en requerimientos.

Programar sigue siendo una parte fundamental del desarrollo de software, pero por sí sola no garantiza una buena solución si se parte de una comprensión incompleta del problema. Por eso, analizar, preguntar y validar antes de diseñar también forma parte del trabajo del ingeniero.

Referencias

● [1] International Institute of Business Analysis. The Business Analysis Standard. Toronto:

International Institute of Business Analysis, 2022.

● [2] International Institute of Business Analysis. IIBA Global Business Analysis Core

Standard. Toronto: International Institute of Business Analysis, 2021.

● [3] Bragilovski, Maxim, Ashley T. van Can, Fabiano Dalpiaz, y Arnon Sturm.

"Leveraging Machines to Derive Domain Models from User Stories." Requirements

Engineering 30, no. 2 (2025).

ARTÍCULO · ESTUDIANTE

De la visión al código: estrategias para una gestión efectiva de requerimientos

Autores: Elian Angel Fernando Reyes Yac; Ramiro Agustín Téllez Carcuz

Palabras clave: Arquitectura de software, ingeniería de requerimientos, requerimientos funcionales, requerimientos no funcionales.

Introducción

La ingeniería de software moderna enfrenta el enorme desafío de traducir complejas expectativas humanas en especificaciones sistémicas que resulten verificables. Este artículo analiza la ingeniería de requerimientos para el plasmado de los requerimientos del usuario, se clasifican los distintos tipos de requerimientos existentes, así como, siete funciones en la ingeniería de requerimientos propuestas por Pressman para la elicitación de requerimientos.

Se analiza el uso de maquetas visuales para la validación de requerimientos debido a que puede llegar a ser una herramienta importante para el entendimiento del sistema y la clarificación de requerimientos, esto también previene la implementación errónea de requerimientos. De esta

manera, se busca establecer un marco de trabajo que optimice la comunicación entre los equipos de desarrollo y los usuarios finales.

Desarrollo

La ingeniería de requerimientos se centra en todas las actividades referentes a identificar y documentar las necesidades de los clientes. Para este fin aplica principios científicos y técnicas para desarrollar, comunicar y gestionar los requerimientos del usuario (Medina Cruz, Pineda Ballesteros, and Téllez Acuña, 2019).

La International Institute of Business Analysis define los requisitos como: “Los Requerimientos

Incluyen, pero no están limitados a, a condiciones del pasado, presente o futuro o capacidades en una empresa y descripciones de la estructura organizacional, roles procesos, políticas, reglas, e información del sistema. Un requerimiento debe describir el estado actual o futuro de cualquier aspecto de la empresa” (Wong Durand, 2017). Según Pressman (WONG DURAND, 2017), la ingeniería de requisitos comprende siete funciones que permiten identificar, definir, validar y gestionar las necesidades de un proyecto de software. Estas funciones, que abarcan desde el inicio del proyecto hasta la gestión de los requisitos, se presentan en la Figura 1.

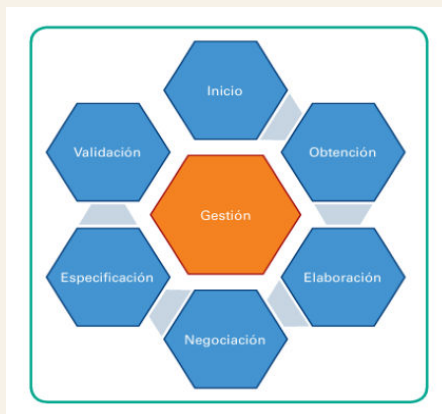


Figura 1. Funciones de la Ingeniería de Requisitos según Pressman. Fuente: Wong Durand (2017).

Como ejemplo, en el posible desarrollo de una herramienta educativa para el aprendizaje de

Código SQL Oracle, primero se identificarían las necesidades de los estudiantes y docentes, determinando qué problemas se busca resolver. Posteriormente, se establecerían los límites del proyecto y los requisitos principales, como la ejecución de consultas SQL, la práctica de normalización y la elaboración de diagramas entidad-relación.

Con toda esta información se analizarían aspectos como el tiempo y los recursos disponibles, se elaborarían prototipos para comprobar que la propuesta responde a las necesidades planteadas y,

finalmente, los requisitos serían validados y registrados para controlar los cambios que puedan surgir durante el desarrollo.

El uso complementario de maquetas visuales sirve como un potente mecanismo de validación e iteración constante entre los ingenieros y los clientes. Al utilizar maquetas funcionales de manera

temprana se clarifica el entendimiento del sistema. Esta técnica permite prevenir la implementación errónea de funciones no evaluadas correctamente. Las consideraciones previas y el diseño de pantallas son fundamentales para garantizar que la

CONTINUACIÓN · ARTÍCULO

De la visión al código: estrategias para una gestión ef

herramienta digital proporciona una usabilidad sobresaliente y satisfactoria. Un software bien diseñado técnica y visualmente reduce en gran medida las frustraciones en el uso cotidiano.

Conclusiones

La transformación efectiva de necesidades humanas en especificaciones técnicas requiere la aplicación rigurosa de metodologías de ingeniería, diseñadas específicamente para minimizar la incertidumbre y reducir drásticamente la frustración del usuario final durante la operación del

sistema. Es necesario realizar un análisis documental exhaustivo, integrando múltiples perspectivas orientadas al usuario, lo cual facilita la construcción de un panorama integral.

Convertir las necesidades subjetivas de los usuarios en métricas medibles y cuantificables resulta

un paso absolutamente crucial para lograr definir el alcance de los proyectos informáticos de una manera realista, coherente y profesional. En definitiva, los patrones de interacción y la experiencia del usuario constituyen los pilares fundamentales que moldean tanto la solución general como la escalabilidad futura de la arquitectura, garantizando así el éxito comercial.

Referencias

Arciniegas, José L., Verónica Fernández, Amparo Hormiga, Aleyda Tulande, Fernando A

Urbano, and César A Collazos. 2009. "Proceso de requerimiento y análisis para la

definición de la arquitectura desde la perspectiva de usabilidad para el desarrollo de

aplicaciones en la Web." *Avances en Sistemas e Informática* 6, no. 2 (Septiembre):

205-210. <http://www.redalyc.org/articulo.oa?id=133113598023>.

Medina Cruz, Javier, Eliécer Pineda Ballesteros, y Freddy Reynaldo Téllez Acuña. 2019.

"Requerimientos de software: prototipado, software heredado y análisis de documentos."

Ingeniería y Desarrollo 37 (2): 327-345. <https://doi.org/10.14482/inde.37.2.1053>.

Wong Durand, Sandra. 2017. *Análisis y requerimientos de software*. Huancayo, Perú:

Universidad Continental.

<https://hopelchen.tecnm.mx/principal/sylabus/fpdb/recursos/r138017.PDF>.

ARTÍCULO · ESTUDIANTE

La arquitectura que nadie pidió: cuando las decisiones técnicas se alejan de las necesidades del usuario

Autores: María Patricia Serrano Ramírez; Adler Alejandro Pérez Asensio

Palabras clave: Arquitectura de software, Decisiones arquitectónicas, Requisitos de software, Sobreingeniería, Atributos de calidad.

Introducción

¿De qué sirve construir una arquitectura técnicamente impecable si nunca fue necesaria para

resolver el problema? En el desarrollo de software, una decisión arquitectónica debe surgir siempre de las necesidades y las restricciones que definen el sistema. Ignorar esto puede convertir lo que era una solución sencilla en una compleja generando un aumento de costos y esfuerzo.

Por ello, los requisitos funcionales y los atributos de calidad deben construir la base para determinar qué arquitectura es adecuada para el sistema. El verdadero desafío al diseñar un sistema no es generar la arquitectura más sofisticada sino lograr justificar aquella que responda

de manera efectiva al problema.

De la necesidad a la decisión arquitectónica

¿Qué pasaría si la arquitectura de un sistema fuera la respuesta a una pregunta que nunca se hizo? En el desarrollo de software, el adoptar tecnologías antes de lograr comprender correctamente el problema hace que una solución sencilla se vuelva una estructura innecesariamente compleja. Por ello, las decisiones deben ser fundamentadas por los requisitos

del sistema y no por la popularidad de las tecnologías.

Tanto los requisitos funcionales como los atributos de calidad, rendimiento, seguridad, disponibilidad y escalabilidad deben orientar las decisiones de arquitectura (Lago, Avgeriou, and

Hilliard 2010). Cada componente que se incorpora al sistema debe responder a una necesidad concreta y debe aportar un beneficio que se pueda justificar.

Cuando la arquitectura supera al problema

La Sobre Ingeniería ocurre cuando la solución técnica que se plantea es mucho más compleja del problema a resolver. Un ejemplo es adoptar una arquitectura distribuida para una aplicación

sencilla. Aunque estas decisiones pueden parecer preparación futura, genera una consecuencia medible al chocar con los atributos de calidad, disparando costos de mantenimiento, pruebas y dificultades que no existían sin un beneficio que lo justifique (Kazman, Klein, and Clements 2000).

Esto no significa que la complejidad sea negativa ya que hay sistemas que si lo necesitan como los sistemas que procesan datos sensibles o las plataformas con miles de usuarios. En estos casos, estas arquitecturas generan beneficios reales. El problema surge cuando una solución compleja se adopta sin tener un requisito que la justifique. La complejidad deja de ser una inversión cuando no se responde a las necesidades reales del sistema convirtiéndose en una complejidad accidental, introducida por tendencias, imitaciones o expectativas poco realistas.

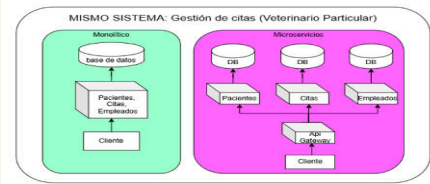
Un mismo problema, diferentes soluciones

Bien, para verlo de mejor manera se plantea el caso: “Desarrollo de sistema para la gestión de citas de un veterinario particular”, contemplando registrar usuarios, agendar cita, consultar disponibilidad, cancelar cita, entre otras. Para darle solución al problema veremos dos posibilidades, una de ella utilizando un monolítico y una arquitectura basada en microservicios.

CONTINUACIÓN · ARTICULO

La arquitectura que nadie pidió: cuando las decisiones

mejor manera se plantea el caso: “Desarrollo de sistema particular”, contemplando registrar usuarios, agendar cita, entre otras. Para darle solución al problema ella utilizando un monolítico y una arquitectura basada



ción de una arquitectura monolítica y una arquitectura basada en microservicios para un mismo sistema de gestión. Fuente: Elaboración propia.

nta la Figura 1, una arquitectura basada en microser

Figura 1. Comparación de una arquitectura monolítica y una arquitectura basada en microservicios para un mismo sistema de gestión. Fuente: elaboración propia.

Tal como lo presenta la Figura 1, una arquitectura basada en microservicios representaría aproximadamente el triple de despliegues a mantener, pues se observan 3 servicios, que precisamente en el caso planteado, cuentan con un sistema de persistencia cada uno, frente a una aplicación global y una base de datos que representaría un monolítico. Ojo, que no automáticamente significa que está mal, pero estamos hablando de un “Sistema para la gestión de citas de un veterinario particular”, no hay ningún driver, como alta disponibilidad o escalamiento masivo que justifique la implementación de un proyecto de tal magnitud.

Conclusión

Para diseñar una arquitectura es importante tener en cuenta que no se construye un sistema para relucir conocimientos técnicos, o para dar a conocer habilidades. Esos dos factores son esenciales, pero no son la prioridad; la prioridad en todo momento es buscar la mejor solución al problema inicial y satisfacer las necesidades del usuario. Por supuesto, la propuesta tiene que tener fundamento técnico y tiene que ser hecha a la medida, y en esta parte es donde se tiene que demostrar cuán capacitado está un arquitecto, porque esto no va de suponer que estilo arquitectónico es mejor valorado, o más difícil de implementar, consiste en buscar una solución apta. Basar la arquitectura en la definición de “architecture drivers” es una práctica indispensable, pues estos ponen restricciones, límites, y alcances que condicionan la elaboración y planificación de un proyecto, esta práctica ayuda a que el producto final sea una solución real.

Referencias bibliográficas

1. Lago, Patricia, Paris Avgeriou, and Rich Hilliard. 2010. Guest Editors’ Introduction: Software Architecture: Framing Stakeholders’ Concerns, IEEE Software 27 (6). <https://pure.rug.nl/ws/files/2632678/2010IEEESoftwareLago.pdf>.

2. Kazman, Rick, Mark Klein, and Paul Clements. 2000. ATAM: Method for Architecture Evaluation, CMU/SEI-2000-TR-004. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. https://www.sei.cmu.edu/documents/629/2000_005_001_13706.pdf.

3. Gonzalez, Rodrigo Alejandro, Sergio Agustín Giménez, Numa Roy Molina Pualuk, and Rodrigo Zalazar. 2024. “¿Son Los Microservicios La Mejor opción? Una evaluación De Su Eficacia Y Eficiencia Frente a Los Monolitos”. JAIIO, Jornadas Argentinas De Informática 10 (5). <https://revistas.unlp.edu.ar/JAIIO/article/view/17980>.

ARTÍCULO · ESTUDIANTE

Reaccionar a tiempo: la diferencia entre ajustar un análisis y rehacer la arquitectura

Autores: José Luis Espinoza Jolón; José David Góngora Olmedo

Palabras clave: Análisis funcional, Arquitectura de software, Historias de usuario, Atributos de calidad, Detección temprana.

Introducción

En todos los proyectos de software surge la tentación de avanzar lo más rápido que se pueda al diseño y desarrollo del software, dejando la parte de análisis de requerimientos como un paso más o solo por “compromiso”. Pero un malentendido de negocio que no se detecta a tiempo no desaparece, sino que se traspa a la arquitectura. Lo que se sentía como tiempo ahorrado termina siendo el origen de un problema más costoso de resolver.

Este artículo surge de una idea simple, pero algo ignorada: ajustar un análisis funcional cuesta minutos, en cambio, rehacer una arquitectura ya construida puede costar semanas de trabajo.

Teniendo en cuenta esa diferencia, se busca entender por qué el costo de un error crece con el paso del tiempo y demostrar que lo importante no es un análisis perfecto, sino la capacidad de reaccionar a tiempo.

Desarrollo

Por qué el costo de un error crece con el tiempo

El costo de corregir un error de requerimientos aumenta drásticamente mientras más tiempo pasa desde el momento en el que se comete y hasta que se detecta (Boehm 1981). Un malentendido detectado en análisis se corrige de inmediato; el mismo malentendido, detectado ya con la arquitectura construida, puede implicar rediseñar componentes completos.

Aun así, esto no se debe tomar como una regla absoluta. Las proporciones exactas dependen del tamaño y alcance del proyecto. Estas también indican que fijar los requerimientos de manera muy anticipada para evitar cambios a última hora puede traer consecuencias, una de ellas: el construir de manera precisa algo que el usuario ni siquiera necesitaba (Boehm y Basili 2001). Esto lleva a considerar que lo relevante no es evitar los cambios lo más que se pueda, sino desarrollar la habilidad de detectar y corregir los problemas lo más rápido posible, antes de que la arquitectura sea construida sobre un error.

Un caso simple es: un cliente pide ‘un sistema que gestione pedidos’ sin aclarar si incluye facturación o notificaciones. Detectar esa ambigüedad en análisis cuesta una pregunta; detectarla con el módulo ya construido implica rediseñarlo por completo.

Ilustración 1 Gráfica ilustrativa de la tendencia en el costo de corregir un error según la etapa del proyecto



Ilustración 1. Gráfica ilustrativa de la tendencia en el costo de corregir un error según la etapa del proyecto. Fuente: elaboración propia con Excel, basada en el concepto de Boehm (1981).

Cómo se ve esto en la práctica

Para entender de mejor manera como crece el costo de un error, debemos analizar qué sucede cuando un error pasa desapercibido durante el análisis. Si se detecta una mala interpretación a tiempo, la corrección se limita a un documento, sin embargo, si escala al nivel de arquitectura, una simple aclaración se convierte en semanas de trabajo desperdiciado.

Un arreglo sencillo puede realizarse en las historias de usuario y los casos de uso. Un ejemplo ayuda a ver esta diferencia con claridad. En un sistema de aprobación de pagos, el requisito original decía algo así: "el sistema debe aprobar los pagos de forma rápida y segura". Esta redacción no dice nada verificable: ¿qué tan rápido es rápido?, ¿qué reglas definen que un pago sea seguro? Durante el análisis, el equipo lo rehace como: "aprobar automáticamente los pagos menores a Q5,000 en un máximo de 3 segundos, siempre que el usuario tenga historial sin incidencias en los últimos 6 meses; los pagos mayores requieren validación manual". Como señalan Wiegers y Beatty (2013), modificar un requisito en papel evita que

CONTINUACIÓN · ARTÍCULO

Reaccionar a tiempo: la diferencia entre ajustar un aná

los errores se transmitan al código, lo que permite realizar ajustes sin agotar el presupuesto.

Errores de diseño que afectan atributos de calidad (escalabilidad o modelado de datos) detectados en producción requieren intervenciones severas, como reestructurar la base de

datos o rediseñar componentes clave. La prioridad técnica debe centrarse en la detección temprana mediante análisis de requerimientos, no en la corrección reactiva.

Este mismo criterio se aplica al uso de IA en el análisis funcional, ya que puede acelerar la redacción de una historia de usuario o sugerir un primer requisito, pero no puede decidir qué es ambiguo ni interpretar el contexto del negocio. Esta validación sigue siendo responsabilidad del analista.

Conclusión

Lo que separa a un proyecto exitoso de uno costoso no es contar con un análisis perfecto, sino la capacidad de reaccionar a tiempo ante un error. Como muestran Boehm (1981) y

Boehm y Basili (2001), el costo de una corrección no depende de la complejidad del error, sino del momento en que se detecta: mientras siga siendo una idea en un documento, ajustarla cuesta minutos; una vez que se traslada a la arquitectura, cuesta semanas de rediseño.

El análisis funcional no es un trámite que retrasa el desarrollo, sino la herramienta que permite corregir el rumbo mientras todavía es barato hacerlo (Wiegiers y Beatty 2013). Una arquitectura sólida no depende de la suerte ni de programar rápido, sino de cuestionar cada requisito ambiguo antes de construir sobre él. La diferencia entre ajustar y rehacer se decide mucho antes de escribir la primera línea de código.

Referencias

Boehm, Barry W. 1981. *Software Engineering Economics*. New Jersey: Prentice-Hall.

Boehm, Barry, y Victor R. Basili. 2001. "Software Defect Reduction Top 10 List".

Computer 34, n. 1: 135-137.
<https://ieeexplore.ieee.org/document/962984>

Wiegiers, Karl, y Joy Beatty. 2013. *Software Requirements*. 3.^a ed. Redmond, WA:

Microsoft Press

ARTÍCULO · ESTUDIANTE

Arquitecto de la torre de marfil (ITA): dificultades del papel a la implementación

Autores: Marco Fernando Cruz Mendoza; Erick Estuardo Muñoz Escalante

Palabras clave: Arquitectura de software, torre de marfil, erosión arquitectónica, desarrollo de software, toma de decisiones, ITA.

Introducción:

La arquitectura de software permite definir la estructura de un sistema y orientar las decisiones necesarias para su desarrollo. Sin embargo, cuando estas se toman ignorando la realidad del proyecto y con poca comunicación con el equipo encargado de implementarlas, puede surgir una brecha entre el diseño y la construcción. Esta situación se puede relacionar con la problemática del arquitecto de la torre de marfil.

El presente artículo explora las dificultades que surgen cuando las decisiones arquitectónicas se alejan de las condiciones reales del desarrollo. También busca destacar la importancia de la comunicación y colaboración entre arquitectos y desarrolladores de software, para favorecer la retroalimentación y considerar las diferentes perspectivas sobre el proyecto.

Desarrollo:

La arquitectura de software cumple una función importante en el desarrollo de software, debido a que establece la estructura y la comunicación entre los distintos componentes que conforman un sistema. Sin embargo, surge la siguiente interrogante: ¿Qué pasaría si el arquitecto encargado se apeg a reglas y estándares «irreales» para establecer el modelo de desarrollo que se utilizará en la construcción del software?

La problemática de la torre de marfil es una metáfora utilizada para hacer referencia a una posición de aislamiento, asociada con figuras que desempeñan un rol de liderazgo. Como se ilustra en la Figura 1, este aislamiento puede favorecer una cadena de desconexiones que culmina en la erosión arquitectónica del sistema. En contraste, la comunicación constante y la colaboración entre arquitectos y desarrolladores pueden favorecer una arquitectura más adaptable y alineada con las necesidades reales del proyecto.

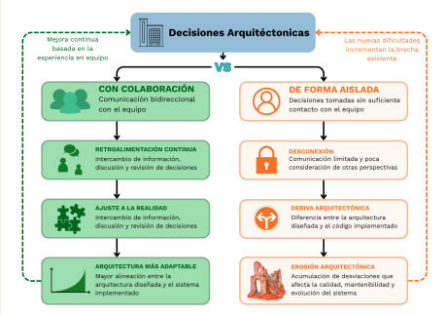


Figura 1. Comparación de los efectos de la toma de decisiones aislada y colaborativa en la arquitectura de software. Fuente: elaboración propia.

arquitectura de software. Fuente: Elaboración propia.

Debido al rol de liderazgo del arquitecto de software, puede presentarse la problemática de la torre de marfil cuando sus decisiones se alejan de las herramientas, los procesos y las necesidades reales del equipo. Esta desconexión puede dificultar la implementación de la arquitectura propuesta y favorecer diferencias entre el diseño y el sistema construido.

Un ejemplo puede presentarse cuando un arquitecto decide implementar una solución cloud ready basada en microservicios, Redis y Kubernetes en un sistema on-premise altamente restrictivo. Desde el punto de vista administrativo, el diagrama de arquitectura puede parecer lógico e impecable. Sin embargo, para el equipo de desarrollo la problemática no se resuelve, sino que se agrava, debido a que el diseño asume conectividad permanente y alta disponibilidad en la nube, ignorando las restricciones del negocio y la naturaleza offline de los sistemas de facturación.

En este caso, la disponibilidad y la escalabilidad constituyen atributos de calidad que orientan la decisión arquitectónica, pero implican un trade-off frente a las restricciones de conectividad y las necesidades operativas del sistema. Entonces, ¿cómo pueden atenuarse los efectos negativos asociados con la problemática de la torre de marfil en la arquitectura de software? Según Emmanuel Valverde Ramos, desde una perspectiva de equipo, es posible

CONTINUACIÓN · ARTÍCULO

Arquitecto de la torre de marfil (ITA): dificultades de

reducir estos efectos mediante las siguientes estrategias:

- Mantenerlo sencillo: utilizar explicaciones claras y sencillas.
- Reuniones casuales: realizar reuniones en un contexto de intercambio de ideas.
- Registrar tiempos entre ciclos: obtener datos relevantes sobre deficiencias.

Conclusiones:

La problemática del arquitecto de torre de marfil demuestra que una arquitectura no debe evaluarse únicamente por su calidad en el diseño, sino también por su capacidad para adaptarse a las condiciones reales de su implementación. Cuando no existe una alineación entre los objetivos de ambas perspectivas se pueden generar diferencias entre la estructura planeada y el producto final, dando como resultado problemas como la erosión arquitectónica. Como recomendación, debe existir una revisión y comunicación frecuente entre el arquitecto y los desarrolladores de software, de manera que la toma de decisiones se pueda contrastar con la realidad del proyecto. Es importante encontrar un equilibrio entre ambas perspectivas, de modo que la visión del arquitecto tampoco se vea limitada, para que se puedan incorporar soluciones con capacidad de evolución, pero sin agregar complejidad innecesaria.

Referencias bibliográficas:

Baabad, A., Zulzalil, H., Hassan, S., & Baharom, S. Characterizing the Architectural Erosion

Metrics: A Systematic Mapping Study. IEEE Access, 10 (2022): 22915-22940.

<https://doi.org/10.1109/ACCESS.2022.3150847>

Uzun, B., & Tekinerdogan, B. Detecting deviations in the code using architecture view-based

drift analysis. Computer Standards & Interfaces, 87 (2024): 103774.

<https://doi.org/10.1016/j.csi.2023.103774>.

Alami, A., Krancher, O., & Paasivaara, M. The journey to technical excellence in agile

software development. Information and Software Technology, 150 (2022): 106959.

<https://doi.org/10.1016/j.infsof.2022.106959>

Valverde Ramos, E. The Ivory Tower Architect. The Tech Lead Journal, Substack (2024).

<https://emmanuelvalverderamos.substack.com/p/the-ivory-tower-architect>

ARTÍCULO · ESTUDIANTE

Comunicación efectiva entre analistas, arquitectos y programadores

Autores: Jens Jeremy Pablo Sosof; Kevin Estuardo Sotoj García

Palabras clave: Comunicación, Requisitos, Código, Arquitectura de Software

Introducción

En el mundo moderno, los sistemas de software se han convertido en la columna vertebral de muchos sectores. Procesos manuales muy complejos, hoy se resuelven en menos tiempo por medio de plataformas digitales. Esto es posible por la ingeniería de software, ya que dividió el desarrollo en roles especializados: analista, arquitecto y programador.

La comunicación entre estos tres roles no es un aspecto secundario ya que determina directamente el éxito o fracaso del desarrollo de software. La inteligencia artificial puede automatizar parte de la traducción entre necesidades del negocio y las especificaciones técnicas. No obstante, la validación final debe permanecer en manos del analista humano.

Desarrollo

La comunicación en el desarrollo de software traduce necesidades humanas en código funcional a través de especificaciones y diagramas. Estas herramientas codifican las reglas de

negocio y relaciones lógicas que el equipo de programación debe implementar.¹

Una comunicación clara produce software robusto, mantenible y alineado con el presupuesto y tiempos del negocio. Este flujo entre roles opera en tres fases:

1. Especificación y Modelado: Levantamiento de requisitos y reglas de negocio detalladas por parte del analista para sentar las bases funcionales del sistema.
2. Diseño de Arquitectura y Contratos: Traducción de los requisitos funcionales en diagramas estructurales, patrones de diseño y contratos de API que guían la fase de codificación.
3. Codificación, Evaluación y Ajuste: Implementación del código, retroalimentación técnica continua sobre las restricciones reales encontradas en el desarrollo, y ajuste dinámico de las especificaciones funcionales y de diseño.

Según The Chaos Report de Standish Group y estudios como el de Rashaad R. Randall, las

brechas de comunicación entre los equipos de negocio hacen que el 80% de los proyectos excedan presupuestos o plazos de entrega.³ Este dato es relevante en equipos pequeños, donde una sola persona suele asumir a la vez el rol de analista y de arquitecto. En esos casos, pueden surgir ambigüedades que pasan desapercibidas hasta etapas avanzadas del desarrollo.

Ventajas de una comunicación fluida	Inconvenientes de la brecha comunicativa
Alineación con el negocio: el software satisface necesidades y objetivos.	Desalineación y retrabajo: se construyen características innecesarias por interpretar mal los requisitos.
Mayor eficiencia: reduce ciclos tardíos de corrección y desperdicio.	Acumulación de deuda técnica: atajos de código degradan el sistema.
Mantenibilidad y escalabilidad: el código sigue el diseño planificado.	Desviación arquitectónica: el código final se aleja de la estructura diseñada.

Tabla 1. Ventajas vs. inconvenientes de la comunicación. Fuente: elaboración propia, 2026, a partir de las fuentes bibliográficas consultadas.

tabla comparativa que resume las principales ventajas de una comunicación fluida frente a los inconvenientes derivados de una brecha comunicativa.

La interacción humana aún presenta graves desafíos.² Destacan la alucinación de requisitos como asumir funcionalidades inexistentes, los outputs técnicos incoherentes, APIs que no respetan los contratos de diseño y la pérdida de traducción por documentación ambigua o la falta de un lenguaje de dominio común. Si bien las herramientas de inteligencia artificial actuales pueden guiar, ayudar o realizar parte del trabajo de esta traducción, estas no reemplazan la validación humana que debe realizar un analista. Por ejemplo, un modelo de inteligencia artificial puede producir una especificación muy completa pero fuera del alcance

real del proyecto. La validación humana es el control final del software desarrollado.

Conclusiones

La comunicación ha sido y sigue siendo un tema de discusión dentro de la ingeniería de software desde hace décadas. La creciente complejidad de los ecosistemas en la nube, la adopción de microservicios y herramientas de inteligencia artificial, han impulsado la

CONTINUACIÓN · ARTÍCULO

Comunicación efectiva entre analistas, arquitectos y pr

necesidad de una alineación sin precedentes entre analistas, arquitectos y programadores. Una comunicación estructurada y ágil ofrece enormes beneficios en términos de productividad y calidad para las organizaciones. Aunque coordinar la visión de negocio, el diseño técnico y la ejecución del código presenta retos humanos y organizacionales reales, las empresas exitosas continúan explorando cómo las nuevas herramientas de inteligencia de software, el modelado

automático y las plataformas de colaboración en tiempo real pueden refinar sus flujos de trabajo internos y enriquecer sus productos finales.

Referencias

1. Pedro Pablo González Pérez e Ismael Soto Galindo, "Modelado, arquitectura y comunicación en el desarrollo de sistemas de software: un enfoque práctico" (libro de texto, Universidad Autónoma Metropolitana, 2023), consultado el 17 de agosto de 2026, http://ilitia.cua.uam.mx:8080/jspui/bitstream/123456789/1160/1/Modelado%20arquitectura%20y%20comunicacion%20en%20el%20desarrollo%20de%20sistemas%20de%20software_%20un%20enfoque%20practico-VF.pdf

2. José Cruz Estefanía Almanza, "Propuesta para reducir la brecha de comunicación mediante pruebas de software basadas en comportamiento ISO/IEC 29110" (tesis de maestría, Universidad Autónoma de Querétaro, 2026), consultado el 20 de agosto de 2026, <https://ri-ng.uaq.mx/bitstream/123456789/13079/1/IFMAC-253822.pdf>

3. Rashaad R. Randall, "Lost in Translation: The Cost of Communication Gaps", en Insights on Software Engineering & Leadership, consultado el 20 de agosto de 2026, <https://rashaadrandall.com/insights/lost-in-translation-the-steep-cost-of-software-communication-gap>



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

SECCIÓN 3

Seguridad, despliegue y operación

¿Quién protege el sistema y responde cuando llega a producción?



ARTÍCULOS DESTACADOS

- El especialista en seguridad antes y después de la IA
- Seguridad desde el diseño y responsabilidad compartida
- Despliegues automatizados con supervisión humana
- Código y configuraciones generadas por IA
- Incidentes, observabilidad y recuperación



ENTREVISTA • PROFESIONAL

Ing. Marlon Francisco Orellana López

Coordinador del Área de Software - Docente

Automatización de procesos - Analítica de datos - Gestión tecnológica



TEMA

Automatizar no significa dejar de decidir

[Ver la entrevista](#)

PRESENTACIÓN

Marlon Francisco Orellana López es ingeniero en sistemas, colegiado desde 2006, con alrededor de veinte años de docencia en la Facultad de Ingeniería. Coordina el Área de Software, participa en la Escuela de Posgrados y ha desarrollado su carrera privada en analítica de datos, administración de proyectos tecnológicos y automatización de procesos.

ANTES DE LA IA GENERATIVA

Antes de esta revolución de inteligencia artificial y agentes, ¿cómo era la automatización y qué procesos repetitivos dependían completamente de una persona?

Al mirar hacia los años noventa, Orellana recuerda una transición progresiva. Muchas tareas que antes vivían en papel comenzaron a digitalizarse: reportes tabulados, registros, cálculos y gráficos pasaron a apoyarse en computadoras y hojas de cálculo. Eso redujo imprecisiones y permitió manejar más información, pero la intervención humana seguía siendo fuerte. Durante años la automatización fue, en realidad, semiautomatización: la máquina ayudaba, pero una persona continuaba ejecutando, verificando o trasladando buena parte del trabajo.

El salto llegó con la estandarización de datos y procesos, el uso de la nube, la seguridad y el crecimiento del volumen de información. Con millones de registros y tiempos de respuesta más exigentes, automatizar dejó de ser comodidad y pasó a ser una necesidad para reducir costos, errores y tiempo. Allí aparecen procesos donde la intervención humana puede ser mínima, siempre que el riesgo sea bajo y el resultado pueda supervisarse.

TRANSICIÓN

Usted ubica alrededor de 2015 un cambio importante. ¿Qué impulsó ese salto hacia una automatización mucho más agresiva?

Para Orellana, el cambio no fue únicamente tecnológico. También fue económico y cultural: las organizaciones comenzaron a necesitar decisiones más rápidas y las personas empezaron a valorar de otra manera su tiempo. Si una actividad puede repetirse automáticamente, dedicar horas humanas a ejecutarla una y otra vez deja de ser una justificación suficiente del trabajo.

El objetivo pasa entonces de operar a decidir. Una persona puede dejar que el sistema procese lo repetitivo y utilizar ese tiempo para revisar resultados, detectar excepciones y tomar decisiones estratégicas. Esa presión coincide con un cambio generacional y con una economía cada vez más veloz. Automatizar, en ese contexto, significa liberar capacidad humana para tareas de mayor valor.

ENTREVISTA • CONTINUACIÓN

Ing. Marlon Francisco Orellana López

Continuación

CRITERIO

Cuando identifica un proceso repetitivo, ¿cómo decide si realmente vale la pena automatizarlo o si debería mantenerse con intervención humana?

La primera distinción debe hacerse entre ejecución y decisión. Una actividad operativa, estandarizada, repetitiva y susceptible a errores humanos es una buena candidata para automatización, especialmente si consume tiempo, incrementa costos o retrasa otras decisiones. Pero automatizar la ejecución no significa automatizar el criterio.

Orellana insiste en que el filtro humano no debería desaparecer. Un sistema puede procesar información con enorme rapidez y entregar un resultado, pero alguien debe revisar si ese resultado tiene sentido dentro del contexto de la organización. Mientras más difícil sea revertir una acción o mayor sea su impacto, más importante se vuelve conservar una instancia humana de revisión.

RESPONSABILIDAD

¿Qué decisiones sí pueden quedar completamente en manos de una automatización y cuáles deberían seguir siendo humanas?

Su respuesta establece una frontera basada en riesgo. Las decisiones sin intervención humana deberían limitarse a escenarios de bajo impacto, donde un error sea pequeño, controlable, reversible y no comprometa vidas, patrimonio, reputación ni variables críticas. Allí la autonomía puede ser razonable porque la falla es aceptable y manejable.

La situación cambia cuando hay consecuencias económicas importantes, efectos sobre personas o reputación institucional. En esos casos la decisión debe permanecer en manos humanas. Orellana pone como ejemplo una llamada de emergencia: una máquina puede clasificar palabras y señales, pero comprender plenamente violencia, accidente o peligro exige una riqueza contextual que todavía vuelve indispensable al ser humano.

CRITERIO

¿Qué cambia cuando una automatización deja de seguir reglas predefinidas y comienza a interpretar información o decidir qué acción ejecutar?

Allí aparece uno de los riesgos centrales de los agentes actuales. En una automatización tradicional se busca un comportamiento determinístico: dadas ciertas condiciones, el sistema debe llegar a un final conocido y aceptable. Cuando un agente explora caminos, interpreta información y elige acciones por sí mismo, deja de ser evidente qué peso asignará a cada variable y qué considerará un acierto o un desacierto.

La respuesta de Orellana es establecer límites. Un agente necesita fronteras claras, tolerancias y condiciones que acoten su comportamiento. Si permanece dentro de esos límites, la autonomía puede ser útil; si se le deja el camino completamente abierto, el equipo pierde capacidad de predecir las consecuencias.

RESPONSABILIDAD

Si un agente o una IA comete un error, ¿el sesgo pertenece a la creación o al creador que definió sus límites?

Orellana sitúa la responsabilidad en quien diseña y permite el margen de acción. Si el creador estableció límites, tolerancias y condiciones de error, también definió hasta dónde podía desviarse el sistema. Cuando un agente actúa dentro de un rango autorizado, la responsabilidad no puede trasladarse simplemente a la herramienta: ese comportamiento fue posible porque alguien lo permitió.

El riesgo aumenta cuando se entrega autonomía sin fronteras suficientes. Un sistema abierto puede producir resultados impredecibles o utilizarse incluso con fines destructivos. Los límites, por tanto, no son una restricción accidental: son parte esencial de la ingeniería. Automatizar puede significar delegar ejecución; no significa delegar responsabilidad.

CIERRE

Automatizar tiene sentido cuando desplaza esfuerzo operativo hacia trabajo estratégico. El reto ya no es demostrar que una máquina puede ejecutar una tarea, sino definir qué puede hacer, hasta dónde puede llegar y cuándo debe devolver la decisión a una persona.

ARTÍCULO · ESTUDIANTE

La seguridad no se agrega al final: una responsabilidad compartida desde el diseño

Autores: Daniel Eduardo Izas Marroquin; Alan Rodrigo Pamal De León

Palabras clave: Seguridad desde el diseño, responsabilidad compartida, desarrollo seguro, gestión de riesgos, ciclo de vida del software

Introducción

La seguridad del software ya no puede considerarse una actividad que se realiza únicamente

después de programar un sistema. Las vulnerabilidades pueden originarse desde la definición

de los requerimientos, la selección de la arquitectura o la configuración inicial del producto.

Por ello, la seguridad desde el diseño propone identificar riesgos y establecer controles desde las primeras decisiones del proyecto, en lugar de corregir los problemas en producción.

Este enfoque requiere reconocer que la protección del sistema no depende únicamente del especialista en seguridad. Analistas, arquitectos, programadores, encargados de pruebas y operadores toman decisiones que pueden aumentar o reducir los riesgos. La seguridad debe convertirse en una responsabilidad compartida durante todas las etapas del desarrollo.

Desarrollo

Cambio de paradigma

En la ingeniería de software, la seguridad suele revisarse al final, poco antes del despliegue.

Esta práctica resulta insuficiente ante amenazas cada vez más sofisticadas, cuya creación también puede verse facilitada por la inteligencia artificial.

Tener un software seguro es una necesidad que se debe abordar desde su planeación, y no como un agregado extra. Es necesario aplicar un enfoque proactivo, y no reactivo, en la seguridad para evitar las pérdidas de tiempo y dinero que pueden surgir por la necesidad de aplicar un parche en un sistema que ya se encuentra en fases avanzadas de creación.

La responsabilidad compartida

Durante el desarrollo de un software, la seguridad no es algo que sea abordado únicamente por un especialista. Por el contrario, es una cultura que debe aplicarse durante todo el ciclo de vida del software por todos los involucrados en su creación. La seguridad comienza desde

que el analista toma las necesidades y las convierte en requerimientos técnicos. Se definen las

distintas reglas de negocio y las restricciones al acceso a la información desde el primer momento; luego, con esta información, se toman decisiones sobre la estructura del software.

Con la implementación de los asistentes de inteligencia artificial y la generación automática de código, el rol del programador cambia a dirigir y validar el código generado. Utilizar dicho código sin revisarlo genera vulnerabilidades, cuya culpa recae sobre el programador. Por lo tanto, el programador es responsable de leer, probar y validar la seguridad de cada parte del código durante la creación.

El ciclo finaliza con el paso a producción. Aunque se cuente con despliegues automatizados, se requiere de una supervisión constante para revisar permisos, configuraciones, credenciales y secretos que puedan estar accesibles para todos.

Estos son para todos.

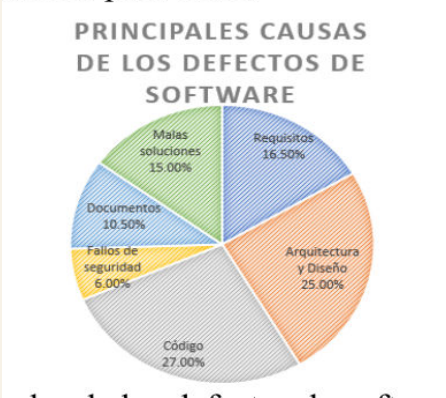


Figura 1. Causas principales de los defectos de software. Fuente: Jones (2018).

Monitoreo y continuidad

La seguridad tampoco termina cuando el sistema llega a producción. Es necesario monitorear registros, configuraciones, accesos y comportamientos anómalos para detectar incidentes oportunamente. La información obtenida debe utilizarse para corregir defectos, actualizar los

CONTINUACIÓN · ARTÍCULO

La seguridad no se agrega al final: una responsabilidad

controles y evitar que los mismos problemas vuelvan a presentarse. Además, deben existir copias de seguridad, procedimientos de recuperación y responsables definidos.

Conclusión

La seguridad debe integrarse desde el inicio, pues muchas decisiones que protegen un sistema se toman antes de escribir el código. Considerarla desde los requerimientos y la arquitectura reduce vulnerabilidades y evita correcciones costosas. Esta responsabilidad corresponde a todos los participantes del ciclo de vida del software. La inteligencia artificial facilita la generación de código, pero sus resultados requieren revisión, pruebas y supervisión humana.

Después del despliegue, el monitoreo continuo ayuda a detectar comportamientos anómalos, corregir defectos y responder ante incidentes. Las copias de seguridad y los procedimientos de recuperación también mantienen la continuidad del servicio. Por ello, la seguridad desde el diseño es necesaria para desarrollar sistemas confiables y protegerlos durante su operación.

Referencias

Jones, Capers. Applied Software Measurement: Global Analysis of Productivity and Quality.

CRC Press, 2018.

National Institute of Standards and Technology (NIST). The NIST Cybersecurity Framework

(CSF) 2.0. NIST Cybersecurity White Paper (CSWP) 29: U.S. Department of Commerce,

2024. <https://doi.org/10.6028/NIST.CSWP.29>.

National Institute of Standards and Technology (NIST). Secure Software Development

Practices for Generative AI and Dual-Use Foundation Models. NIST SP 800-218A. 2024.

<https://doi.org/10.6028/NIST.SP.800-218A>.

¿Cómo evitar que la responsabilidad compartida se diluya en equipos de desarrollo que aplican seguridad desde el diseño?

Autores: Marcos Geovani Barrios López; Josue Daniel Chávez Portillo

Palabras clave: Seguridad desde el diseño, responsabilidad compartida, desarrollo de software, trazabilidad

Introducción

La seguridad de un sistema no comienza cuando el software está terminado, sino desde las primeras decisiones sobre análisis y diseño. Estas decisiones influyen en los riesgos y controles del sistema

a lo largo del desarrollo. Por eso, los involucrados en el proceso de construcción del sistema deben participar para que la seguridad sea una responsabilidad compartida desde el principio y así la

seguridad del sistema no recaiga en una sola persona.

La participación de todo el equipo no garantiza que cada decisión de seguridad tenga una persona responsable en particular. Si las tareas no se asignan claramente, especialmente cuando se utiliza

IA como herramienta de apoyo, la responsabilidad puede diluirse hasta el punto en que nadie sea responsable. Por lo tanto, debemos identificar las brechas en la responsabilidad y remediarlas antes de continuar con el desarrollo.

Desarrollo

La distribución de la seguridad entre múltiples personas solo funcionará si todos entienden qué hacer y hasta qué punto tienen autoridad para tomar decisiones. Debido a la mala comunicación de la distribución, ciertos riesgos pueden pasar desapercibidos porque todos creen que alguien más

se hará cargo de ellos. Por lo tanto, NIST CSF 2.0 establece que las funciones, responsabilidades y autoridades de seguridad deben ser definidas y comunicadas adecuadamente para poder responsabilizar a las personas.¹

Esto no implica que todos los roles de seguridad se le asignen a una sola persona, sino que significa la asignación de tareas específicas. El analista identificará las amenazas y los requerimientos; el arquitecto diseñará el control; el programador lo implementará; y el revisor lo verificará. En el

1 Pascoe, Quinn y Scarfone, The NIST Cybersecurity Framework, 17. artículo actual se discute esta asignación desde las siguientes cuatro perspectivas: técnica, la cual

es responsable de ejecutar y verificar el control; organizativa, la cual es responsable de asignar las tareas; ética, la cual es responsable de analizar el impacto de las decisiones; y legal, la cual es responsable de verificar los requerimientos legales. NIST SSDF ayuda en la asignación de tareas.²

Según OWASP SAMM, el Security Champion es responsable de liderar a su equipo, pero no de realizar todas las tareas.³

Para evitar cualquier dilución de la responsabilidad, proponemos una responsabilidad rastreable,

la cual consta de tres componentes: la parte responsable, el revisor y la evidencia. La parte responsable lleva a cabo la acción; el revisor asegura su cumplimiento; y la evidencia permite probar el resultado. Si falta cualquiera de estos componentes, existe un vacío, el cual debería ser

llenado. El NIST SSDF respalda nuestro enfoque al sugerir que se cree evidencia y que se registren las decisiones durante el proceso de desarrollo.⁴

No es suficiente identificar la falta de responsabilidad cuando el proyecto puede continuar sin

llenar el vacío. En este sentido, proponemos puntos de control antes de la finalización de cada actividad o fase. Si falta una parte responsable, un revisor o una prueba de que la tarea se ha completado, esta deberá quedar pendiente. El NIST SSDF respalda este proceso de revisión y documentación al recomendar controles y el registro de las decisiones durante el desarrollo.⁵ La

Elemento	Preguntas de control	Condición para continuar
Parte responsable	¿Quién realiza la tarea o toma la decisión?	Existe una persona asignada.
Revisor	¿Quién revisa el resultado?	La revisión se llevó a cabo y fue registrada.
Evidencia	¿Existe alguna evidencia de esto?	Existen pruebas o registros.
Estado del control	¿Se han cumplido los tres elementos?	Se avanza únicamente cuando todo esté completado.

Tabla 1. Elementos de verificación en un punto de control. Fuente: elaboración propia.

Este principio también es válido si la inteligencia artificial es utilizada por el grupo como un elemento de apoyo. Por ejemplo, si la inteligencia artificial escribe el código de un módulo de

CONTINUACIÓN · ARTÍCULO

¿Cómo evitar que la responsabilidad compartida se diluya

autenticación, entonces el desarrollador tiene la responsabilidad técnica de verificar la seguridad de este fragmento de código; el líder tiene la responsabilidad organizativa de coordinar su revisión;

el equipo analiza el impacto potencial en el usuario desde la perspectiva ética; mientras que el oficial de cumplimiento revisa los temas legales pertinentes. La recomendación solo será aceptada si hay una persona responsable, un revisor y evidencia de las pruebas realizadas.

Conclusión

Al delegarse las responsabilidades a ciertas personas, será importante mantener la responsabilidad compartida. La participación de todos los miembros del grupo debería ir de la mano con asignaciones claras para evitar acciones sin responsable. Las perspectivas técnica, organizativa, ética y legal nos ayudan a identificar quiénes participan en cada decisión y cómo deben responder.

La identificación de vacíos en las responsabilidades nos ayuda a llenarlos antes de que surja algún

problema. Por lo tanto, cada decisión debe tener una persona responsable, un revisor y evidencia.

Los puntos de control permiten verificar estos requisitos. Este enfoque también se aplica cuando la inteligencia artificial se utiliza como ayuda, pues sus sugerencias deben de ser analizadas de manera crítica y aprobadas.

Referencias bibliográficas

OWASP Foundation. "SAMM Agile Guidance."

OWASP Software Assurance Maturity Model.

Acceso el 20 de agosto de 2026

<https://owasp-samm.org/docs/guides/agile/>.

Pascoe, Cherilyn, Stephen Quinn y Karen Scarfone. The NIST Cybersecurity Framework (CSF)

2.0. NIST Cybersecurity White Paper 29.

Gaithersburg, MD: National Institute of Standards and

Technology, 2024.

<https://doi.org/10.6028/NIST.CSWP.29>.

Scarfone, Karen, Murugiah Souppaya y Donna Dodson. Secure Software Development Framework

(SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST

Special Publication 800-218. Gaithersburg, MD: National Institute of Standards and Technology,

2022. <https://doi.org/10.6028/NIST.SP.800-218>.

ARTÍCULO · ESTUDIANTE

Compila hoy, rompe mañana: los problemas de compilar lo que la IA escribe

Autores: Kevin Josue Santos Salazar; Moisés Iván Daniel Alvarado García

Palabras clave: compilación, mantenibilidad, inestabilidad, despliegue continuo, logs.

Introducción

El código generado por inteligencia artificial ya no es novedad. Actualmente los agentes de IA generan una gran parte de los commits diarios en proyectos reales y repositorios de todo tipo, desde herramientas pequeñas hasta sistemas con miles de usuarios. Y con eso, el debate de la calidad del código sigue detenido en el mismo punto, si el código compila y funciona al entregarse.

Este enfoque ignora la etapa más costosa, que es el mantenimiento a largo plazo. Un programa se escribe una vez, pero se mantiene durante años y la mantenibilidad no se demuestra simplemente haciendo un par de tests que pasan en verde. A partir de un estudio sobre repositorios reales y de un informe anual de la industria, este artículo analiza qué ocurre con ese código meses después de haberse integrado.

Desarrollo

Hacer las cosas a la carrera no sale bien. En programación, arreglar algo rápido para entregarlo hoy genera problemas que alguien tendrá que resolver mañana. La inteligencia artificial no inventó este problema, pero sí tiene una capacidad impresionante para acumular este tipo de dificultades y aumentar el desorden que se genera.

El reporte DORA de 2025 reveló que los equipos terminan tareas más rápido, pero también incrementa la inestabilidad en la entrega de software. El código se genera tan rápido que los pipelines de integración y despliegue de muchas empresas no han evolucionado lo suficiente para procesar grandes volúmenes de cambios de manera segura. Como resultado, aumentan los fallos en los despliegues y la necesidad de rehacer el trabajo.

Al revisar 623 millones de modificaciones de código entre 2023 y 2026 que fueron los años en que la IA llegó a hacer más commits, el estudio de GitClear mostró que la calidad se cayó en todos los frentes:

Señal estructural medida Variación

Llamadas a funciones entre archivos
-35%

Líneas movidas por refactorización
-70%

Cambios sobre código con más de doce meses de antigüedad -74%

Bloques de código duplicados +81%

Señal estructural medida	Variación
Llamadas a funciones entre archivos	-35%
Líneas movidas por refactorización	-70%
Cambios sobre código con más de doce meses de antigüedad	-74%
Bloques de código duplicados	+81%

Tabla 1. Variación de señales de mantenibilidad, 2023-2026. Datos de GitClear.

Los resultados muestran que el problema no está solamente en cuánto código puede producirse, sino en lo que ocurre después de generarlo. La reducción de la refactorización y el aumento del código duplicado sugieren que se dedica menos esfuerzo a reorganizar y mejorar lo existente.

Los datos de GitClear sobre menor refactorización y mayor duplicación muestran que parte del trabajo que se evita al inicio puede trasladarse al mantenimiento posterior. Cuando dichas soluciones rápidas se acumulan, terminan convirtiéndose básicamente en deuda técnica que posteriormente deberá resolverse.

Además, la IA crea la ilusión de corrección pero por dentro trae problemas escondidos, los desarrolladores se enfrentan a problemas que van desde funcionalidades que no sirven o no hacen nada y el borrado silencioso de pruebas unitarias, hasta la caída de servicios o todo el sistema.

Debido a esto, el trabajo del desarrollador no desaparece, sino que cambia en vez de escribir código, ahora dedica más tiempo a corregir errores y validar logs. Por esa falta de confianza se entiende por qué los programadores dicen que no le tienen fe a la calidad del código hecho por IA.

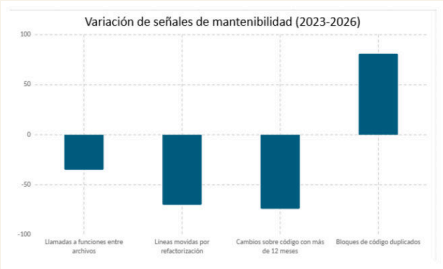


Figura 1. Variación de señales de mantenibilidad (2023-2026). Fuente: elaboración propia con base en datos de GitClear (2026).

La figura 1 lo deja claro, ninguna barra queda cerca de cero, lo que señala un cambio de hábito sostenido y no una variación menor. Al mismo tiempo, creció 47 % la costumbre de esconder errores con parches rápidos, es decir, silenciando los fallos en lugar de arreglarlos de raíz.

Conclusiones

La inteligencia artificial ha beneficiado en muchos aspectos, su uso para generar código puede ayudar en gran medida en el desarrollo de las funcionalidades con mayor rapidez, pero que un programa funcione, en su entrega, no necesariamente significa que la calidad sea la óptima según los requerimientos y menos a largo plazo, la IA se ha caracterizado por solucionar rápido. Esto de hecho puede incurrir en que el ahorro del tiempo inicial, luego se deba invertir en corregir errores. Por esta razón debe ser una herramienta de apoyo pero no un reemplazo de la revisión de un desarrollador. El beneficio real de la IA dependerá no solo de cuánto código se pueda producir, sino la mantenibilidad que puede llegar a tener en el tiempo.

Esto permite mantener abierta la discusión sobre si el ahorro de tiempo que ofrece la inteligencia artificial justifica las dificultades que pueden aparecer después.

Referencias bibliográficas

GitClear. "The Maintainability Gap: 2026 AI Code Quality Research". GitClear, 2026. https://www.gitclear.com/the_ai_code_quality_maintainability_gap

Google Cloud y DORA. State of AI-assisted Software Development 2025. Google, 2025. [https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.p df](https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.pdf).

Perry, Neil, et al. "Do Users Write More Insecure Code with AI Assistants?" ACM, 2023. <https://doi.org/10.1145/3576915.3623157>.



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

SECCIÓN 4

Programación e inteligencia computacional

¿Cómo cambia el oficio de programar cuando parte del código puede generarse automáticamente?

ARTÍCULOS DESTACADOS



El programador antes y después de los asistentes de IA



Vibe coding: crear sin comprender completamente



Comprender, revisar y defender el código generado



Algoritmos genéticos y creatividad computacional



Del programador que escribe al programador que dirige y valida



ENTREVISTA · PROFESIONAL

TEMA

Ing. Luis Fernando Cajas

Desarrollo de software y coordinación técnica



Programar con IA: del código al criterio

[▶ Ver la entrevista](#)

PRESENTACIÓN

¿Desde qué experiencia observa hoy el cambio en la programación?

Luis Fernando Cajas se define ante todo como desarrollador. Acumula cerca de diez años de experiencia en software y actualmente coordina un equipo orientado a análisis de datos, business intelligence y presentación de información. Ese recorrido le permite mirar la IA desde dos perspectivas: como alguien formado en la depuración y la documentación tradicional, y como responsable de equipos que hoy pueden automatizar gran parte del trabajo mecánico.

ANTES

Antes de los asistentes de IA, ¿cómo investigaba, depuraba y refactorizaba código?

El proceso dependía mucho más de documentación oficial, buscadores, foros técnicos, Stack Overflow y videos especializados. Cuando aparecía un error había que reproducirlo, entender exactamente qué ocurría, buscar casos similares, formular hipótesis y probar soluciones una por una. Era más lento, pero obligaba a conocer el código. Para Luis, la IA redujo drásticamente el tiempo de búsqueda; lo que no eliminó fue la necesidad de comprender aquello que se está intentando resolver.

ANTES

¿Qué diferenciaba a un buen programador?

No era escribir más rápido, sino resolver mejor. Había que descomponer una necesidad en problemas pequeños, reconocer restricciones y seleccionar una solución adecuada. Estructuras de datos, algoritmos, bases de datos, arquitectura, sistemas operativos, redes y patrones de diseño eran -y siguen siendo- fundamentos. A ello se sumaba una habilidad esencial: depurar. Seguir la ejecución, plantear hipótesis y comprobar dónde se rompe un sistema forma un criterio que ninguna respuesta generada sustituye automáticamente.

TRANSICIÓN

¿Cuál fue una de las primeras tareas que cambió con estas herramientas?

En su trabajo reciente con grandes volúmenes de información, Luis vio el cambio en tareas mecánicas como la generación de consultas SQL. La IA puede acelerar la sintaxis, pero el valor profesional aparece después: revisar índices, pensar en rendimiento, anticipar problemas y optimizar la consulta. El aprendizaje de la transición fue claro: no se traslada el problema completo a la herramienta. La calidad del resultado depende del contexto entregado y de la capacidad de revisar lo que devuelve.

TRANSICIÓN

¿Qué riesgos encuentra en el llamado vibe coding?

Le parece una ventaja enorme porque reduce la barrera de entrada y permite obtener prototipos con rapidez. El riesgo aparece cuando se confunde que una aplicación funcione con que esté bien construida. Puede ejecutar una tarea y aun así contener fallos de seguridad, arquitectura, rendimiento, manejo de errores o mantenimiento. El problema no es utilizar lenguaje natural para construir; es hacerlo sin comprender las decisiones que terminaron dentro del sistema.

ENTREVISTA • CONTINUACIÓN

Ing. Luis Fernando Cajas

Continuación

DESPUÉS

Si la IA produce código que funciona pero el desarrollador no puede explicarlo, ¿debería integrarse?

Para Luis, no de forma directa. Que compile es apenas una condición. Hay que saber qué hace, por qué lo hace, cuáles son sus límites y cómo reaccionará en escenarios no previstos. Si el desarrollador no puede explicar el fragmento tampoco puede garantizar mantenimiento ni responder ante una falla. Antes de integrarlo deberían existir revisión técnica, pruebas de seguridad y arquitectura y, cuando la criticidad lo amerite, una segunda persona que revise la solución.

DESPUÉS

¿Cómo debería revisarse el rendimiento y la mantenibilidad del código generado?

La automatización no elimina áreas de revisión; puede volverlas más importantes. Equipos de QA, revisión técnica y pruebas ayudan a confirmar que la velocidad de generación no se convierta en velocidad para introducir errores. En sistemas con grandes volúmenes de información, por ejemplo, una consulta correcta puede seguir siendo costosa o difícil de mantener. La revisión debe mirar más allá de la salida inmediata y considerar cómo se comportará el componente cuando el sistema crezca.

DESPUÉS

¿Está cambiando el rol del programador?

Sí, hacia un nivel de abstracción mayor. Parte del tiempo que antes se dedicaba a escribir cada línea puede trasladarse a definir restricciones, diseñar soluciones, integrar componentes y validar resultados. Eso no vuelve prescindible al programador; vuelve más importante su criterio. La IA puede hacer a un buen profesional más productivo, pero también permite que un mal criterio produzca errores a una velocidad mucho mayor.

RESPONSABILIDAD

¿Por qué sigue siendo humana la definición de una buena solución?

Porque optimizar no equivale a comprender el propósito. Si una función de aptitud, una restricción o un objetivo están mal definidos, el algoritmo puede encontrar una solución técnicamente óptima para una pregunta equivocada. La máquina puede buscar, generar y optimizar; alguien debe establecer qué significa éxito, qué límites no pueden cruzarse y qué impacto es aceptable. Automatizar la búsqueda de una solución no automatiza la definición del problema.

CIERRE

¿Qué debería dominar un programador aunque una herramienta pueda generar casi cualquier implementación?

Lógica, fundamentos y capacidad de programar sin depender por completo del asistente. Luis insiste en que comprender algoritmos, estructuras, arquitectura y depuración sigue siendo necesario. Sobre esa base pueden añadirse agentes, automatización, inteligencia computacional o nuevas plataformas. La diferencia profesional no estará en negarse a usar IA, sino en poder utilizarla sin entregar a la herramienta el criterio con el que se juzga el sistema construido.

IDEA CENTRAL

Automatizar la búsqueda de una solución no significa automatizar la definición del problema.

¿Reemplazo o transformación? El impacto de la inteligencia artificial en la programación

Autores: Eduardo Antonio Reyes Pineda; Adriana Lucia Ojeda Rivas

Palabras clave: Inteligencia artificial, programación, asistentes de código, responsabilidad profesional, revisión de código

Introducción:

Durante años ser programador significó dominar varios lenguajes y resolver problemas escribiendo cada línea de código manualmente. Sin embargo, con la llegada de los asistentes de IA, esta forma de trabajar ha cambiado rápidamente. Muchos desarrolladores comenzaron a ver estas herramientas como una amenaza para sus empleos, especialmente porque la IA puede generar código sin que una persona tenga que escribirlo desde cero.

A pesar de estas preocupaciones, la situación es más compleja que un reemplazo. Este artículo analiza cómo la IA ha cambiado el trabajo diario de los desarrolladores, dando mayor importancia a revisar, corregir y validar el código. También se analiza cómo estos cambios afectan de forma diferente a los perfiles junior y senior. A partir de estudios recientes, se busca determinar si la IA está reemplazando a los programadores o transformando su trabajo.

Desarrollo:

Tras la llegada de la IA, muchos programadores temieron perder sus empleos. Sin embargo, el código generado aún presenta errores, por lo que estas herramientas requieren supervisión humana [1]. Un estudio reciente indica que el 75% de desarrolladores continúa revisando manualmente el código generado por IA, ya que no se fían al 100% del código que la misma IA les genera y recurren a revisar minuciosamente cada fragmento de código antes de integrarlo en sus proyectos [2]. Esto demuestra que, aunque la IA puede generar código, los desarrolladores todavía necesitan revisarlo antes de integrarlo en sus proyectos.

Casado Martínez y Clarisó Viladrosa comparan este rol con el de un director de orquesta: quien corrige, ajusta y garantiza el resultado final [1]. Así la IA permite a los desarrolladores centrarse más en revisar, validar y decidir sobre el código. Sin embargo, la transformación afecta de forma distinta a senior y junior. Los senior se enfocan más en arquitectura y

revisión, mientras que, según Tobore, un solo ingeniero puede realizar el trabajo de un equipo completo con IA. Esto plantea dudas sobre las oportunidades para quienes recién comienzan. Los desarrolladores junior sí se están viendo afectados, ya que las ofertas de empleo disminuyeron un 49%, y quienes utilizan la IA sin revisar críticamente el código presentan una comprensión hasta un 17% menor que quienes programan de forma manual [3]. Esto podría explicar en parte la disminución de contrataciones, al exigir mayor solidez técnica.

Sector	Caídas en oferta
Desarrolladores de Software	49%
Científicos de datos junior	29%
Analistas informáticos	22%

Tabla 1. Caída de ofertas de empleo por sector tras la adopción de IA generativa. Fuente: elaboración propia con datos de Cvejic, Rebelo y Seve.

Sector Caídas en oferta

Como se observa en la Tabla 1, la IA redistribuye responsabilidades según la experiencia, haciendo que el criterio y la detección de errores sean más importantes que la velocidad de programación.

Conclusiones:

La inteligencia artificial está transformando la forma en que trabajan los programadores. Su capacidad para generar código reduce el tiempo dedicado a tareas repetitivas. Sin embargo, el código generado todavía necesita revisión y supervisión humana. Por esta razón, revisar, validar y corregir se vuelven habilidades muy importantes. Este cambio también afecta de manera diferente a los desarrolladores junior y senior. Los perfiles senior pueden aprovechar la IA para enfocarse más en arquitectura y decisiones técnicas, mientras los perfiles junior enfrentan mayores dificultades para adquirir experiencia y acceder a empleos. Esto demuestra que la IA no implica necesariamente el reemplazo de los programadores, sino un cambio de responsabilidades y habilidades. Por lo tanto, aprender a utilizar la IA sin depender de ella será clave para los futuros desarrolladores.

Referencias:

[1] Casado Martínez, Carlos, y Ismael Robert Clarisó Viladrosa. "La IA, los programadores

y la nueva revolución del código." UOC, 16 de diciembre de 2025.

<https://blogs.uoc.edu/informatica/es/impacto-ia-profesionales-programacion/>

[2] Tobore. "How AI Is Reshaping Software Development and the Tech Industry in 2026."

Medium, 25 de febrero de 2026.

<https://medium.com/@tobore/how-ai-is-reshaping-software-development-and-the-tech-industry-in-2026-4ec7f7a801df>

[3] Cvejic, Ksenija, Luís Daniel Rebelo, y Raphael Seve. "Working with the Machine: How

AI Is Reshaping Software Developer Careers in 2026." Devoteam, 2026.

<https://www.devoteam.com/expert-view/ai-impact-software-developer-careers-2026/>

ARTÍCULO · ESTUDIANTE

Generado por IA, aprobado por ti: la nueva responsabilidad del programador

Autores: Henry David Quel Santos; Gerardo Leonel Ortiz Tobar

Palabras clave: Código generado por IA, revisión de código, comprensión del código, responsabilidad del desarrollador, asistentes de programación, deuda técnica.

Introducción

Los asistentes de programación basados en inteligencia artificial, como GitHub Copilot,

ChatGPT o Claude, se integraron al flujo diario de miles de desarrolladores y generan funciones o módulos enteros a partir de una simple instrucción en lenguaje natural. Esta rapidez cambió la

forma de escribir software, pero también introdujo una pregunta incómoda: ¿quién responde cuando ese código, escrito por una máquina, falla en producción o expone una vulnerabilidad?

Frente a esta realidad surge un principio que debería guiar a cualquier equipo de desarrollo:

ningún fragmento de código generado automáticamente debe integrarse a un sistema sin que una

persona lo comprenda, lo revise y responda por él. Este artículo analiza por qué comprender, revisar y defender ese código se ha vuelto una competencia tan importante como saber escribirlo, y qué prácticas permiten sostenerla en equipos reales de ingeniería.

Desarrollo

En un experimento controlado, los desarrolladores que usaron Copilot completaron una tarea de programación 55% más rápido que quienes no lo usaron (GitHub 2022). Sin embargo, la velocidad no equivale a confianza: según la Encuesta de Desarrolladores de Stack Overflow, apenas el 43% confía en la precisión de las herramientas de IA, mientras un 31% se declara escéptico (Stack Overflow 2024). Esta brecha es el punto de partida de este artículo: usar un asistente de código no exime de comprenderlo, revisarlo y responder por él.

Comprender un fragmento de código generado significa poder explicar qué hace y qué supuestos asume sobre los datos de entrada. Los asistentes de código no razonan sobre el problema particular del equipo: producen una solución estadísticamente plausible a partir de patrones observados en otros proyectos. Un análisis de más de ciento cincuenta millones de líneas de código encontró que el código descartado o reescrito en las dos semanas siguientes a su creación

se duplicó entre 2021 y 2024, señal de que buena parte de ese código se integra sin comprenderse a fondo (GitClear 2023).

La revisión de código, o code review, sigue siendo el mecanismo más confiable para detectar errores antes de producción, y su importancia crece cuando el autor original es un modelo de lenguaje. No debe limitarse a validar que el código compile: debe verificar que la lógica sea correcta, que no introduzca vulnerabilidades y que sea mantenible por el resto del equipo. Los modelos de lenguaje pueden generar salidas inseguras sin un filtro humano, por lo que conviene

tratar cualquier código producido por IA con el mismo escrutinio que el código de una persona desconocida (OWASP Foundation 2025).

Ningún marco de gestión de riesgos de IA ni norma de la industria exime al desarrollador de responder por el código que publica, sin importar quién o qué lo haya escrito (NIST 2023). En una auditoría, un incidente de seguridad o una revisión legal, la pregunta no es «¿quién lo generó?», sino «¿por qué se aceptó esta implementación?». Defender el código significa poder

justificar cada decisión: por qué se eligió cierta estructura de datos, por qué se manejó así un error o por qué se confió en una dependencia externa. Esa justificación debe sostenerse incluso cuando la sugerencia inicial provino de un asistente de IA.

La figura 1 resume este ciclo de responsabilidad: comprender, revisar y defender no son pasos aislados, sino una práctica continua que se repite cada vez que se acepta una sugerencia generada automáticamente.

CONTINUACIÓN · ARTÍCULO

Generado por IA, aprobado por ti: la nueva responsabili

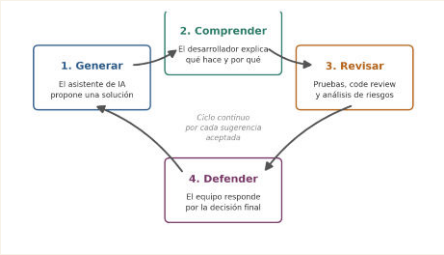


Figura 1. Ciclo de responsabilidad sobre el código generado por IA. Fuente: elaboración propia, con base en OWASP Foundation (2025) y NIST (2023).

Conclusión

Comprender, revisar y defender el código generado por inteligencia artificial no son controles

opcionales: son la condición mínima para adoptar estas herramientas sin heredar una deuda técnica invisible. La velocidad que ofrecen los asistentes de código solo se traduce en valor real cuando existe una persona capaz de explicar, corregir y responder por cada línea que termina en

producción.

El reto para los equipos de ingeniería no es prohibir el uso de estas herramientas, sino construir

la disciplina —revisión sistemática, pruebas y documentación de decisiones— que permita usarlas sin renunciar a la comprensión del sistema que se está construyendo. Esa disciplina, más que la herramienta misma, es lo que distingue a un equipo capaz de sostener su software en el tiempo.

Referencias

GitClear. 2023. "Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality." GitClear. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

GitHub. 2022. "Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness." The GitHub Blog. <https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>

National Institute of Standards and Technology (NIST). 2023. "Artificial Intelligence Risk Management Framework (AI RMF 1.0)." NIST AI 100-1. <https://doi.org/10.6028/NIST.AI.100-1>

OWASP Foundation. 2025. "OWASP Top 10 for LLM Applications 2025." OWASP GenAI

Security Project. Consultado el 11 de septiembre de 2026. <https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/>

Stack Overflow. 2024. "2024 Stack Overflow Developer Survey: AI." Consultado el 11 de septiembre de 2026. <https://survey.stackoverflow.co/2024/ai>

El fin del desarrollo de software tradicional: programadores en la nueva era de la IA generativa

Autores: Fernando José Vicente Velásquez; Alexander Santiago Ceballos Gil

Palabras clave: Inteligencia Artificial Generativa, Ingeniería de software, Evolución del programador, Desarrollador de software, Calidad de software.

Introducción

Desde la llegada de la Inteligencia Artificial generativa han llegado a surgir muchas incógnitas sobre si

muchas profesiones cambiarían o desaparecerían por completo; el desarrollo de software no fue la excepción. Observar a la IA generativa completar tareas que antes requerían de mucho tiempo generó

un fuerte impacto. Afortunadamente los desarrolladores de software no van a desaparecer, pero es

necesario que se lleve a cabo una renovación en cuanto a su formación académica.

Es vital transformar tanto el rol como la formación de los desarrolladores. Anteriormente, su trabajo se

limitaba a la codificación manual; sin embargo, aunque la automatización acelera el desarrollo, la falta

de buenas prácticas genera riesgos en los sistemas. Por ello, el objetivo de este artículo es analizar

cómo debe actualizarse este rol en la nueva era de la IA generativa.

El nuevo rol del programador en el ciclo del desarrollo de software

La formación de nuevos programadores debe adaptarse a priorizar la capacidad crítica de identificación y comprensión de problemas. Es vital que se capaciten en la delegación y orquestación

de recursos técnicos. Esto implica desarrollar un criterio de evaluación ética y de gobernanza.¹ La tarea

de los programadores debe consistir en tener el criterio de identificar requerimientos primordiales.

Fallos en la generación no gobernada

Cuando se utilizan herramientas de IA sin respaldo académico y sin verificación técnica se genera el fenómeno llamado generación no gobernada. 2 Si no se posee control de estos fallos y no se audita el código activamente, se pueden llegar a manifestar los siguientes modos de fallos recurrentes en el

desarrollo.

1 Mamdouh Alenezi, "Educating the Agentic Engineer: Curricula, Collaboration, and Continuous Learning in the

AI Era," arXiv preprint arXiv:2607.29610 (2026)

2 Mamdouh Alenezi, "Specification-Driven Development as the Foundation of AI-Native Enterprise Software Engineering," arXiv preprint arXiv:2607.16680 (2026), 2-3, <https://arxiv.org/html/2607.16680>. Como se expone en la Tabla 1, la dependencia excesiva en la IA sin supervisión humana provoca problemas serios. Se destaca la erosión de la arquitectura y la acumulación de vulnerabilidades.

Modo de fallo	Mecanismo causal
F1: Paradoja productividad/fluibilidad	La rapidez y el flujo incentivan la aceptación del comportamiento observado; La práctica de control de calidad es escasa o ausente; Distancia de corrección entre "puede funcionar" y "se corrige"
F2: Erosión arquitectónica	Generador optimiza el contexto del prompt localizando; Los propietarios del sistema de control transversal son invisibles dentro de una sola generación; Los errores locales se acumulan direccionadamente
F3: Exposición a la seguridad	Generación propaga la vulnerabilidad agravada por una aceptación humana excesiva; Las debilidades llegan a los repositorios de producción
F4: Acumulación de deuda técnica	El razonamiento detrás del código generando nunca se externaliza en artefactos revisables; Entrelazamiento y deriva entre intención e implementación

Tabla 1. Modos de fallo en la generación de código de IA no gobernada. Fuente: adaptado de Mamdouh Alenezi, "Specification-Driven Development as the Foundation of AI-Native Enterprise Software Engineering".

AI-Native Enterprise Software Engineering"

Evidencia empírica de la deuda técnica (velocidad vs. calidad)

Un análisis de la firma de analíticas de código GitClear, evaluó más de 153 millones de líneas de código escritas entre los años 2020 al 2023. Se encontró que, el uso de asistentes de IA se asocia con

un aumento de código duplicado, mientras que la reutilización y el refactorizado de código estructurado disminuye. 3 La productividad aumenta, la mantenibilidad y comprensión decaen.

Otra investigación causal desarrollada por la Universidad Carnegie Mellon evaluó de forma empírica

la adopción de entornos de desarrollo ágenticos. Se determinó que la adopción de agentes de IA genera un incremento significativo pero temporal

en la velocidad de desarrollo de un proyecto. Sin embargo, a largo plazo, provoca un aumento persistente en las advertencias de análisis estático

y en la complejidad estructural del código.⁴ Es vital priorizar la revisión humana para mitigar esta deuda técnica.

CONTINUACIÓN · ARTÍCULO

El fin del desarrollo de software tradicional: programa

Soluciones de gobernanza: el desarrollo basado en especificaciones

La arquitectura de esta solución se consolida en el Modelo de Referencia para la Gobernanza de

Especificaciones. Dado que los modelos de lenguaje generan respuestas basadas en probabilidades, es

3 William Harding y Matthew Kloster, Coding on Copilot: 2023 Data Suggests Downward Pressure on Code

Quality, whitepaper (GitClear, 2024), 5,

https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality. 4 Hao He et al., "Does AI-Assisted Coding Deliver? A Difference-in-Differences Study of Cursor's Impact on Software Projects," arXiv preprint arXiv:2511.04427 (2025), 4, <https://arxiv.org/abs/2511.04427>. necesario controlarlos mediante reglas claras. De esta forma, la generación automática de código por medio de agentes de IA, queda obligada a cumplir estrictamente con tres niveles progresivos de rigor y

control metodológico.5

Conclusiones

Las instituciones de educación superior enfrentan el reto inmediato de reformar radicalmente sus programas de estudio para formar a las futuras generaciones de desarrolladores de software. Los planes

curriculares deben dejar de medir el éxito de los estudiantes por su capacidad por medio de métricas

que solo se centren en la sintaxis del código y es necesario comenzar a evaluar sus capacidades de diseño lógico de requerimientos, auditoría de código y gobernanza de la calidad del software. El

Desarrollo Basado en Especificaciones representa la respuesta metodológica e institucional más sólida

para embridar la fluidez probabilística de la IA. Al estructurar el desarrollo de software alrededor de especificaciones legibles por máquina que sirven como contratos formales, se logra encerrar la naturaleza estocástica de la IA dentro de límites deterministas de verificación y control. Es así como se

consigue la evolución del programador centrándose en un rol evolucionado.

Referencias bibliográficas

Alenezi, Mamdouh. "Educating the Agentic Engineer: Curricula, Collaboration, and Continuous

Learning in the AI Era." arXiv preprint arXiv:2607.29610, 2026. <https://arxiv.org/html/2607.29610>.

---, Mamdouh. "Specification-Driven Development as the Foundation of AI-Native Enterprise

Software Engineering." arXiv preprint arXiv:2607.16680, 2026. <https://arxiv.org/html/2607.16680>.

---, Mamdouh. "The Rise of AI-Native Software Engineering: Implications for Practice, Education, and the Future Workforce." arXiv preprint arXiv:2606.12986, 2026. <https://arxiv.org/abs/2606.12986>.

Harding, William, y Matthew Kloster. Coding on Copilot: 2023 Data Suggests Downward Pressure on

Code Quality. Whitepaper, GitClear, 2024.

https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality.

He, Hao, Courtney Miller, Shyam Agarwal, Christian Kästner, y Bogdan Vasilescu. "Does AI-Assisted

Coding Deliver? A Difference-in-Differences Study of Cursor's Impact on Software Projects." arXiv preprint arXiv:2511.04427, 2025. <https://arxiv.org/abs/2511.04427>.

5 Mamdouh Alenezi, "Educating the Agentic Engineer: Curricula, Collaboration, and Continuous Learning in the AI Era," arXiv preprint arXiv:2607.29610 (2026), 4-5, <https://arxiv.org/html/2607.29610>.



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

SECCIÓN 5

Personas, documentación y continuidad

¿Cómo puede perdurar un sistema cuando cambian sus herramientas y sus responsables?

ARTÍCULOS DESTACADOS



Documentar para otras personas



Transferencia de conocimiento en equipos



El liderazgo técnico antes y después de la IA



Formación de nuevos programadores en la era generativa



Continuidad, mantenimiento y memoria organizacional



ENTREVISTA · PROFESIONAL

TEMA

Ing. Víctor Bladimir Castellanos Albeño

Formación, continuidad y personas



Personas, conocimiento y responsabilidad

▶ [Ver la entrevista](#)

PRESENTACIÓN

Perfil del Ing. Víctor Bladimir Castellanos Albeño

Antes de que comenzara formalmente la entrevista, la conversación ya mostraba una idea importante: la tecnología no vive aislada. Al hablar de un posible centro de datos en Zacapa surgieron ubicación, agua, energía, ruido, comunidades, costos y sostenibilidad. La respuesta de Víctor Castellanos fue prudente: una idea puede parecer beneficiosa en el papel y aun así encontrarse con reacciones humanas que no se pueden predecir. La planificación debe considerar a las personas desde el inicio.

El diálogo previo refleja una relación de formación más que de simple transmisión de contenidos. Víctor Castellanos recuerda alumnos, posibles asesorías de tesis y el deseo de que algo de lo enseñado haya permanecido en quienes pasaron por sus cursos. Esa perspectiva conecta con la continuidad: el conocimiento técnico vale más cuando no queda encerrado en una sola persona, sino cuando puede convertirse en criterio para otros.

FORMACIÓN

¿Qué significa especializarse de verdad?

La conversación sobre ciberseguridad separa con claridad una credencial de una competencia. Tener un título no garantiza dominio práctico. La especialización exige estudiar, experimentar y ser capaz de demostrar que se comprende el área en situaciones reales. Esa exigencia es especialmente relevante en un entorno donde una herramienta puede producir respuestas convincentes: la formación debe permitir distinguir entre saber utilizar una respuesta y saber por qué esa respuesta es correcta.

ENTREVISTA • CONTINUACIÓN

Ing. Victor Castellanos

Continuación

PROYECTOS

¿Qué debería mirar un ingeniero además de la solución técnica?

Viabilidad económica, impacto ambiental y efecto social. El ejemplo del centro de datos conduce rápidamente a preguntas sobre energía solar, uso y tratamiento de agua, localización, inversión y relación con la comunidad. Es una buena síntesis del trabajo profesional: una solución puede ser técnicamente posible y, aun así, requerir decisiones que exceden el código. La ingeniería termina dialogando con presupuesto, territorio y personas.

CONTINUIDAD

¿Cómo se evita que un equipo dependa de una sola persona?

El fragmento disponible no llega a la pregunta formal sobre documentación, pero sí deja un principio compatible con ella: formar a otros. Asesorar, explicar y compartir experiencia reduce la dependencia del individuo. La documentación cumple una función semejante cuando captura decisiones, contexto y conocimiento suficiente para que otra persona pueda continuar. En ambos casos el objetivo es el mismo: que el proyecto sobreviva al cambio de responsables.

CIERRE

¿Qué idea resume mejor el enfoque humano de la conversación?

Una frase aparece de manera directa: «Debes siempre ayudar a las personas». Más allá del tono informal del diálogo previo, esa idea ofrece un cierre coherente para la sección. Herramientas, lenguajes y puestos cambian; la capacidad de crear oportunidades, transferir conocimiento y hacer que otros puedan continuar sigue siendo una responsabilidad humana que ninguna automatización vuelve irrelevante.

EJES PREVISTOS PARA LA ENTREVISTA FORMAL

- ¿Cómo documentaban decisiones, arquitectura, APIs, configuración y errores conocidos antes de los asistentes generativos?
- ¿Qué riesgos aparecen cuando el conocimiento crítico depende únicamente de una persona?
- ¿Qué prácticas de transferencia de conocimiento ayudan a incorporar nuevos integrantes?
- ¿Qué documentación resulta realmente útil meses o años después?
- ¿Qué responsabilidades humanas no deberían desaparecer aunque la IA pueda generar documentación?

La transcripción entregada se interrumpe justo al comenzar la entrevista formal. Se preserva aquí únicamente el diálogo realmente registrado; las preguntas del recuadro provienen de la guía editorial y no se presentan como respuestas del entrevistado.

ARTÍCULO · ESTUDIANTE

Del desarrollador al equipo: conocimiento para un software que perdure

Autores: Josué Nabí Hurtarte Pinto; Christian Alexandro Aragón García

Palabras clave: transferencia de conocimiento, continuidad del software, conocimiento tácito, documentación técnica, factor camión.

Introducción

En los proyectos de software, el repositorio no cuenta toda la historia. Puede contener el código y parte de la documentación, pero muchas decisiones, configuraciones o soluciones quedan en la

experiencia de quienes han trabajado directamente con el sistema. El problema aparece cuando solo

una persona conoce esos detalles y el resto del equipo depende de ella para continuar ciertas tareas.

Compartir este conocimiento ayuda a que el proyecto no quede ligado a integrantes específicos. Para

lograrlo se pueden utilizar revisiones de código, mentorías, documentación, sesiones técnicas y rotación de responsabilidades. Más que conseguir que todos conozcan cada parte del sistema, se busca que el equipo tenga suficiente contexto para mantenerlo y continuar trabajando cuando sus integrantes cambien.

Desarrollo

Cuando el conocimiento depende de una persona

Bjørnson y Dingsøyr diferencian el conocimiento que puede quedar registrado del que se adquiere con la experiencia [1]. En software esto se nota cuando existe documentación y, aún así, todos buscan

al desarrollador que conoce mejor cierto módulo. Configuraciones, errores anteriores o decisiones técnicas pueden quedar fuera de los documentos; el riesgo es perder también el contexto necesario para interpretarlos.

Un concepto relacionado es el factor camión o truck factor, que estima cuántas personas podrían dejar un proyecto antes de que la pérdida de conocimiento lo afecte seriamente. Avelino y sus colaboradores estudiaron este problema en proyectos de software [2]. La Tabla 1 resume el riesgo: no

se busca que todos sepan todo, sino que una tarea crítica pueda continuar si falta quien la realiza normalmente.

Aspecto Hallazgo o aplicación en el equipo

Explícito y tácito El explícito se documenta; el tácito se adquiere principalmente con la experiencia.

Factor camión Mide el riesgo de depender de pocas personas en partes críticas.

Revisión de código Expone a otros integrantes a cambios y módulos que no desarrollaron.

Aspecto	Hallazgo o aplicación en el equipo
Explícito y tácito	El explícito se documenta; el tácito se adquiere principalmente con la experiencia.
Factor camión	Mide el riesgo de depender de pocas personas en partes críticas.
Revisión de código	Expone a otros integrantes a cambios y módulos que no desarrollaron.

Tabla 1. Transferencia de conocimiento. Elaboración propia con base en [1]-[3].

Transferir es más que documentar

Documentar ayuda, pero una guía no conserva todo el contexto detrás de una decisión. Por eso también son útiles las mentorías, sesiones técnicas y revisiones de código. Rigby y Bird muestran

que esta práctica permite conocer el trabajo de otros integrantes y distintas partes del sistema. La

Figura 1 evidencia cómo este intercambio amplía el conocimiento del equipo [3].

Estas prácticas funcionan mejor dentro del trabajo cotidiano. Revisar cambios, registrar decisiones y rotar responsabilidades evita que el conocimiento se concentre en las mismas personas. Una señal de

que la transferencia funciona es que otro integrante pueda atender una falla o modificar un módulo

sin depender de quien lo desarrolló. Así, la experiencia individual se vuelve conocimiento del equipo.

CONTINUACIÓN · ARTÍCULO

Del desarrollador al equipo: conocimiento para un softw

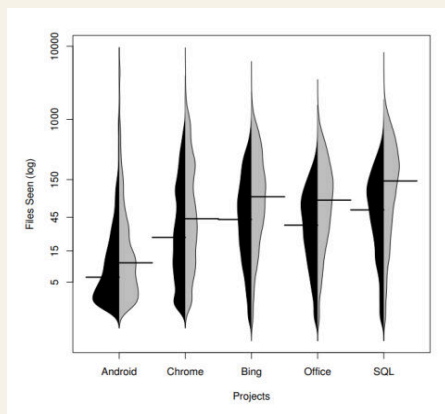


Figura 1. Archivos enviados (izq.) y enviados o revisados (der.). Fuente: Rigby y Bird [3], fig. 8.

Conclusiones

La continuidad del software no depende solamente de conservar su código, sino también del conocimiento que mantiene el equipo para comprenderlo y modificarlo. Cuando decisiones, procesos

o componentes importantes dependen de una sola persona, su ausencia puede convertirse rápidamente en un problema. La documentación ayuda a reducir este riesgo, pero no debería utilizarse como el único medio para compartir conocimiento.

Prácticas como las revisiones de código, mentorías, sesiones técnicas y rotación de responsabilidades permiten que más integrantes comprendan las partes importantes del proyecto.

No es necesario que todos conozcan cada detalle, sino evitar que exista una única persona capaz de continuar una tarea crítica. Compartir conocimiento durante el desarrollo facilita la llegada de nuevos integrantes y permite que la experiencia individual permanezca dentro del equipo.

Bibliografía

[1] Bjørnson, Finn Olav, y Torgeir Dingsøyr. "Knowledge Management in Software Engineering: A

Systematic Review of Studied Concepts, Findings and Research Methods Used." *Information and Software Technology* 50, no. 11 (2008): 1055-1068.

<https://www.sciencedirect.com/science/article/abs/pii/S0950584908000487>

[2] Avelino, Guilherme, Leonardo Passos, Andre Hora, y Marco Tulio Valente. "A Novel Approach

for Estimating Truck Factors." 2016 IEEE 24th International Conference on Program Comprehension

(ICPC). IEEE, 2016. <https://ieeexplore.ieee.org/document/7503718>

[3] Rigby, Peter C., y Christian Bird. "Convergent Contemporary Software Peer Review Practices."

En *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 202-212.

New York: ACM, 2013.

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/rigby2013convergent.pdf>



ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

REVISTA CIENCIAS, SISTEMAS Y TECNOLOGÍA



Conoce más sobre
nuestra revista



✉ revistaecys@usac.edu.gt

CIENCIA
SISTEMAS
SOCIEDAD