Implementación de arquitectura limpia y prácticas DevOps para reducir la deuda técnica en simuladores del portal de contenidos Bancolombia

Sebastián Suárez Ramírez

Trabajo de semestre de industria para optar al título de Ingeniero de sistemas

Asesores
Carlos Andrés Mera Banguero, Ph.D.
Leidy Yoana Román Torres, Esp.

Universidad de Antioquia
Facultad de ingeniería
Departamento de ingeniería de sistemas
Medellín
2024

Cita		Suárez Ramírez [1]
Referencia Estilo IEEE (2020)	[1]	S. Suárez Ramírez, “Implementación de arquitectura limpia y prácticas DevOps para reducir la deuda técnica en simuladores del portal de contenidos Bancolombia”, Semestre de industria, Ingeniería de sistemas, Universidad de Antioquia, Medellín, 2024.

Agradecimientos

Agradezco profundamente a todos mis compañeros de equipo, cuyo apoyo y colaboración fueron esenciales para el desarrollo de mi práctica. En especial, quiero agradecer a mis mentores Andrea Paola Sánchez y Lausnay Rodríguez, quienes me guiaron con su conocimiento y experiencia. También extiendo mi agradecimiento a mis líderes Edison Montoya, Leidy Román y Yohana Monsalve, por su apoyo y por creer en la importancia de este trabajo para la organización. A mi asesor Carlos Mera, por su orientación académica y por compartir su experiencia, la cual fue fundamental para el desarrollo y finalización de este documento.

Agradezco a la empresa Bancolombia por proporcionar los recursos y el entorno necesario para llevar a cabo mis prácticas. Igualmente, a la Universidad de Antioquia por la formación académica y el apoyo institucional recibido durante todo este proceso.

TABLA DE CONTENIDO

LISTA DE TABLA S

LISTA DE FIGURAS

SIGLAS, ACRÓNIMOS Y ABREVIATURAS

CA	Arquitectura limpia
MVC	Modelo vista controlador
AWS	Amazon Web Services
API	Application programming interface
SOAP	Simple object access protocol
CLI	Command line interface
REST	representational state transfer
LTS	Soporte a largo plazo
PO	Dueño de producto
DOD	Definition of done

RESUMEN

Bancolombia S.A. es una institución que brinda productos y servicios financieros en Colombia y otros países de Latinoamérica. Al ser una empresa del sector financiero, Bancolombia cuenta con múltiples servicios de simulación que ayudan a sus clientes, entre otras, a proyectar el valor de las cuotas o a estimar el valor de un préstamo según el valor de la cuota que el cliente puede pagar. Desde un punto de vista técnico, se ha encontrado que la implementación de estos simuladores no cumple con diferentes lineamientos relacionados con Ingeniería de Software y el proceso de DevOps establecidos por la entidad bancaria, lo que resulta en una acumulación de deuda técnica. Es así como, en este trabajo se presenta el proceso de migración de cuatro simuladores de Bancolombia, considerando en dicha migración una implementación basada en arquitectura limpia y adoptando las prácticas DevOps. Lo anterior busca mejorar la escalabilidad, seguridad, mantenibilidad y calidad del software, siguiendo los lineamientos de la organización. En el proceso se planificaron diversas actividades para la migración que incluyen la adecuación del código existente, la comprensión de la lógica del proyecto, la creación de repositorios, la implementación de pruebas y la planificación del despliegue. Se destaca la contribución del proyecto al mantenimiento y adopción de los lineamientos de seguridad e ingeniería de software establecidos por Bancolombia, así como a la reducción de la deuda técnica presente en los sistemas impactados.

Palabras clave — Deuda técnica, arquitectura limpia, DevOps, mantenibilidad, calidad del software, migración de sistemas, ingeniería de software, integración continua, entrega continua, automatización de pruebas.

ABSTRACT

Bancolombia S.A. is an institution that provides financial products and services in Colombia and other Latin American countries. As a company in the financial sector, Bancolombia offers multiple simulation services that help its clients, among other things, to project the value of installments or to estimate the value of a loan based on the installment amount the client can afford. From a technical standpoint, it has been found that the implementation of these simulators does not comply with various guidelines related to Software Engineering and DevOps processes established by the banking entity, resulting in an accumulation of technical debt. Thus, this work presents the migration process of four Bancolombia simulators, considering in this migration an implementation based on clean architecture and adopting DevOps practices. The aim is to improve the scalability, security, maintainability, and quality of the software, following the organization's guidelines. The migration process included various planned activities such as adapting the existing code, understanding the project logic, creating repositories, implementing tests, and planning the deployment. The project's contribution to the maintenance and adoption of Bancolombia's established security and software engineering guidelines, as well as to the reduction of technical debt in the impacted systems, is highlighted.

Keywords — Technical debt, clean architecture, DevOps, maintainability, software quality, system migration, software engineering, continuous integration, continuous delivery, test automation.

I. INTRODUCCIÓN

El portal www.bancolombia.com es una plataforma que permite el acceso a los canales transaccionales, servicios, información y conexiones sociales con usuarios, tanto clientes como no clientes de Bancolombia S.A. [1]. En los contenidos del portal, hay simuladores que permiten a los usuarios explorar algunos de los servicios financieros que ofrece Bancolombia, estas herramientas son útiles para los clientes pues permiten exponer claramente los servicios que el banco tiene a disposición su disposición. No obstante, los desarrollos que respaldan estos simuladores en el *backend* no cumplen con algunos de los lineamientos establecidos por el banco en términos de arquitectura de software y DevOps. Así, este proyecto se enfocó en la migración de estos simuladores hacia una implementación basada en arquitectura limpia, para reducir la deuda técnica presente y alinearse con las mejores prácticas de construcción, despliegue y mantenimiento de software establecidas por la organización.

La arquitectura limpia es un enfoque de diseño de software que separa los elementos de un diseño en capas en forma de anillos concéntricos, donde los componentes están organizados de manera que cada capa depende únicamente de las capas internas. Esta estructura promueve la independencia y la modularidad, permitiendo que el sistema sea independiente de tecnologías y *frameworks* específicos. Además, es altamente *testeable*, ya que las reglas de negocio pueden probarse sin necesidad de otros componentes. También facilita el cambio de elementos como la base de datos, el servidor web u otros componentes externos sin afectar las reglas de negocio, asegurando así la flexibilidad y adaptabilidad del sistema a distintos entornos y tecnologías externas.

Las practicas *DevOps*, por su parte, incluyen principalmente la integración y entrega continuas (CI/CD), entre otras. La integración continua consiste en fusionar frecuentemente los cambios de código en un repositorio central, realizando compilaciones y pruebas automáticas para detectar errores tempranamente. La entrega continua automatiza el despliegue de código en entornos de producción, asegurando que cada cambio esté listo para ser liberado con seguridad y rapidez. La implementación de estas prácticas en cualquier desarrollo de software mejora la

colaboración entre equipos de desarrollo y operaciones, reduce los tiempos de entrega, incrementa la calidad del software y facilita una mayor agilidad y capacidad de respuesta ante cambios, minimizando errores en producción y garantizando un ciclo de desarrollo más eficiente y fiable.

En este proyecto, los lineamientos para migrar los simuladores incluyeron el uso de versiones LTS del lenguaje de programación *Java* y *Spring Boot* 3.0 como *framework* de desarrollo. La implementación siguió los principios de arquitectura limpia y se estructuró en forma de microservicios. Además, se llevaron a cabo pruebas unitarias, de aceptación y de rendimiento, también se utilizaron herramientas como *SonarQube* para la revisión del código. Se crearon repositorios en Azure DevOps y se seleccionaron las bases de datos adecuadas para cada proyecto. Por medio de la migración y el cumplimiento de estas directrices se pretende mejorar la escalabilidad, aumentar la seguridad, mantenibilidad, y demás atributos de calidad de los sistemas impactados.

II. OBJETIVOS

A. Objetivo general

Migrar los simuladores del portal de contenidos de Bancolombia S.A. a una arquitectura limpia a través de la implementación de buenas prácticas en ingeniería de software con el fin de reducir la deuda técnica existente.

B. Objetivos específicos

- Adoptar una metodología ágil en la migración de los simuladores del portal de contenidos de Bancolombia.
- Reescribir el código de los simulares del portal de contenidos Bancolombia siguiendo los principios de arquitectura limpia y las reglas y estándares de código establecidas por Bancolombia.
- Implementar un plan de pruebas unitarias y pruebas de aceptación para detectar y corregir errores y de este modo garantizar la calidad del código en los simuladores migrados.
- Implementar los *pipilenes* de DevOps establecido por Bancolombia para la puesta en producción de los simuladores migrados.

III. MARCO TEÓRICO

Los sistemas informáticos requieren de un mantenimiento y evolución constante que les permita adaptarse a las nuevas necesidades de su entorno y continuar agregando valor a su dueño. Uno de los obstáculos más importantes que se pueden presentar en escenarios de mantenimiento de software es su propensión a la acumulación de "basura" es decir, deficiencias en calidad interna que hacen que sea más difícil de lo que idealmente sería modificar y ampliar aún más el sistema [2].

A este problema común en el desarrollo de software se le conoce como deuda técnica (TD). Este concepto, es una metáfora introducida por Ward Cunningham [2] que hace referencia al costo del retrabajo causado por las decisiones de diseño tomadas que pueden parecer el camino más corto y fácil para sacar adelante el desarrollo [3], pero que establecen un contexto técnico que puede hacer que los cambios futuros sean más costosos y complejos, impactando de manera negativa en la posibilidad de mantener y evolucionar el sistema [4].

Una de las decisiones más importantes que se debe tomar al diseñar un software es la elección de la arquitectura [5], adicionalmente, las decisiones arquitectónicas son una de las principales fuentes de deuda técnica [6]. Bajo este escenario, la implementación de arquitecturas limpias se presenta como una alternativa de diseño de software que ciertamente ayuda a reducir la deuda técnica al promover una base de código limpia y organizada, con menor complejidad [7] y que es fácil de gestionar y actualizar con el tiempo [8], impactando en la mantenibilidad, escalabilidad y pruebas de los sistemas informáticos de la empresa [5].

El concepto de arquitectura limpia fue introducido por Robert C. Martin en [9], y se propone como un enfoque arquitectónico para el diseño de sistemas que separa el sistema en capas, colocando la lógica de negocio o la lógica de dominio en el núcleo de la aplicación y separándolo de su interfaz de usuario y de los mecanismos de almacenamiento de datos [8]. En su propuesta, Martin establece un modelo conceptual, para la separación de responsabilidades con una serie de capas bien definidas; sin embargo, la materialización de dicho modelo puede llevar a distintas implementaciones [10].

El primer componente de la arquitectura limpia propuesta por Martin es la capa de entidades, en esta se encuentran los objetos y estructuras de datos con las que se representan los elementos de negocio que se quieren gestionar [9]; además, al ser el núcleo de la arquitectura aquí se debe definir el comportamiento esperado de las capas superiores a través de interfaces, que son la clave para desacoplar la tecnología del dominio [10]. Luego encontramos los casos de uso, en esta capa se define la lógica del negocio que orquesta el flujo de datos desde y hacia las entidades [9]. La siguiente capa se conoce como capa de infraestructura o de adaptadores, en ella se definen adaptadores de entrada que permiten exponer las capacidades de los casos de uso bajo cierta tecnología concreta (API, SOAP, CLI) y adaptadores de salida como pueden ser el acceso a la persistencia de datos, en los que se implementan las interfaces definidas en la capa de entidades [10]. En la capa más externa de la arquitectura se encuentran los *frameworks* y *drivers* que permiten, por ejemplo, ensamblar los componentes entre las distintas capas y resolver las dependencias [9], [10].

Otro de los elementos cruciales en el desarrollo de software actual es la implementación de prácticas DevOps. A pesar de que no existe una única definición adoptada en la industria o en la academia sobre DevOps, varias definiciones coinciden en que DevOps va en pro de acortar el ciclo de vida del desarrollo de sistemas y proporcionar una entrega de software de alta calidad y confiabilidad [11].

Algunas de las actividades asociadas a la calidad de software en entornos DevOps incluyen la realización de pruebas automatizadas, el manejo de *logs*, el aseguramiento de una alta cobertura de pruebas automatizadas, la implementación de una arquitectura modular y de bajo acoplamiento, la adopción de una estrategia de despliegue continuo [12], pruebas continuas de seguridad [13] y análisis estático de código [14], entre otras.

DevOps posibilita la automatización de procesos, despliegues más frecuentes, experimentación y mejora continua, aumento de la estabilidad y la calidad, y mejoras en la comunicación y colaboración entre equipos [15].

IV. METODOLOGÍA

Este proyecto se desarrolló con una metodología ágil, más precisamente mediante una versión del marco de trabajo SCRUM. En este *framework* el trabajo es llevado a cabo en ciclos de dos semanas denominados *sprints*, en los cuales se realizan diferentes reuniones que facilitan la comunicación, la colaboración y la alineación del equipo de trabajo, asegurando un proceso iterativo eficiente y una entrega continua de valor al cliente.

La metodología inicia con un *backlog*, una lista priorizada de elementos que representan el trabajo a realizar en el proyecto [16]. Este incluye tanto historias de usuario, que describen las necesidades o funcionalidades deseadas desde la perspectiva del usuario final, así como historias habilitadoras, que son aquellas tareas o funcionalidades necesarias para implementar las historias de usuario pero que no tienen un valor directo para el usuario final.

La planificación del *sprint*, o *Sprint Planning*, marca el inicio de cada *sprint* y es una reunión colaborativa en la que el equipo revisa y prioriza el *backlog* del producto con el *Product Owner* (PO). Durante esta reunión se seleccionan los elementos del backlog que serán abordadas en el *sprint*. El propósito de esta reunión es establecer un objetivo claro para el *sprint* y acordar los entregables específicos que se realizarán durante el período [16], [17] .

La reunión diaria, o *daily*, es un evento breve pero crucial en el que los miembros del equipo comparten su progreso desde la última reunión, identifican posibles obstáculos y planifican el trabajo para las próximas 24 horas. Esta ceremonia fomenta la transparencia y la colaboración, manteniendo al equipo alineado y permitiendo la resolución rápida de problemas [16], [17].

Al finalizar el *sprint*, se lleva a cabo su revisión. Aquí, el equipo muestra el trabajo completado al PO y otras partes interesadas. Durante esta sesión, se demuestran las nuevas características desarrolladas y se recibe retroalimentación sobre su funcionalidad, validando que el trabajo realizado cumple con las expectativas del cliente [16], [17].

La retrospectiva del *sprint* es una reunión posterior a la revisión del *sprint* en la que el equipo reflexiona sobre su desempeño durante el *sprint* anterior. Se discuten los aspectos positivos, las áreas de mejora y se identifican acciones para optimizar el proceso de trabajo en el futuro [16], [17]. Además, a lo largo del *sprint*, se llevan a cabo sesiones de refinamiento, para preparar el *backlog* del producto para futuros *sprints*. Durante estas sesiones, se revisan y discuten las historias de usuario y las tareas pendientes, clarificando requisitos y priorizando el *backlog* [16].

El trabajo realizado para consolidar las migraciones de los microservicios en los *sprints* a lo largo de este proyecto comenzó con una exploración del *frontend* y de la documentación disponible del aplicativo a migrar. También se realizó un reconocimiento del código base existente para identificar las entidades, la lógica de negocio y los puntos de entrada, así como las librerías y herramientas utilizadas. Con la información recolectada, se procedió a la ejecución del proyecto en un ambiente local y se verificó su funcionamiento de acuerdo con lo identificado en la exploración del componente *frontend* en su versión de producción.

Una vez que se logró ejecutar el aplicativo de manera correcta en el ambiente local, se creó el repositorio en *Azure DevOps* para alojar el nuevo código, de este modo se inició el proceso de migración. Para hacer esto, Bancolombia cuenta con un *plugin* de código abierto que permite generar, con muy poca configuración, la estructura inicial del proyecto bajo la arquitectura limpia. La estructura generada incluye la configuración necesaria para un proyecto multimódulo de Java con *Spring Boot*, donde las diferentes piezas que conforman la arquitectura limpia adoptada por Bancolombia (dominio, casos de uso, infraestructura, aplicación) corresponden a los módulos del proyecto y están debidamente configuradas con reglas que impiden las dependencias entre módulos que violen los principios de la arquitectura. Por ejemplo, no es posible hacer referencia a elementos de infraestructura desde el dominio; además, se restringe el uso de dependencias a librerías externas que puedan implicar un acoplamiento del dominio a determinadas herramientas.

Luego de definir la estructura inicial, se crearon los modelos y adaptadores de las entidades de dominio y su comportamiento. Este proceso puede realizarse manualmente, creando los archivos y directorios necesarios, o a través del *plugin* mencionado.

Posteriormente se codificó la lógica de negocio mediante los casos de uso; en los casos de uso se utilizan las abstracciones creadas en el dominio (adaptadores), y se define la lógica de negocio del sistema. Para el caso de los simuladores migrados, se realizaron cálculos relacionados con tasas, tarifas, cuotas y demás información relevante. Esta lógica de negocio, bajo la arquitectura inicial de los simuladores, se encuentra en los servicios, pero allí puede estar "contaminada" con detalles de implementación que complican la modificación y mantenimiento del código. Por otro lado, bajo la arquitectura limpia establecida y desde el dominio, se crean abstracciones genéricas para estos detalles de implementación, a los cuáles posteriormente se les define su comportamiento en los componentes de infraestructura del aplicativo. Para el caso de los simuladores migrados, estos componentes incluyen *dynamoDB* como base de datos, colas de mensajería, gestores de secretos y librerías internas de Bancolombia para realizar auditoría de las peticiones y llevar a cabo los cálculos requeridos por el servicio a implementar.

Durante la codificación de estos componentes de software se implementaron pruebas unitarias para garantizar el funcionamiento aislado de cada pieza de código. Para esto se utilizó el *framework* de *Mockito*, además de las utilidades de pruebas proporcionadas por *Spring Boot*. También se utilizó *Jacoco* para consolidar un informe con la cobertura de código en los diferentes archivos y módulos del proyecto.

Una vez finalizada la fase de codificación y pruebas unitarias, se procedió a la automatización de las pruebas de aceptación. Mientras que en las pruebas unitarias se buscó probar unidades de código de bajo nivel, como métodos y funciones, las pruebas de aceptación se integran a un nivel superior entre la lógica empresarial y la interfaz de usuario [18]. Estas pruebas son realizadas mediante el *framework* de *Karate*, el cual provee un entorno completo para la automatización de este tipo de pruebas. Estas pruebas son de gran importancia puesto que permiten determinar si la pieza de software creada satisface o no todos los requerimientos del negocio [19].

Después, se ajustaron los archivos de configuración para el *pipeline* de *build*. Tras la ejecución de este, se generaron y versionaron los artefactos que serán llevados a los ambientes de desarrollo, certificación o calidad y, finalmente a producción. Los pasos que se realizan dentro de este *pipeline* comprenden el análisis estático de código con *Sonar*, el análisis de vulnerabilidades,

la comprobación de la ejecución satisfactoria de todos los *test* unitarios y la verificación del cumplimiento los lineamientos internos. Si no se ejecutan correctamente alguno de estos pasos, el *pipeline* falla, hasta que se corrijan las brechas detectadas.

Una vez se ejecuta correctamente este *pipeline*, se habilita la posibilidad de hacer uso de la plataforma *SonarQube* para visualizar los hallazgos detectados en el análisis de código, esta información es un insumo clave que se usó para iterar nuevamente sobre el código y corregir o mejorar en la medida de lo posible las métricas que allí se analizaron.

A continuación, se configura el *pipeline* de *release*, el cual realiza el proceso de despliegue de la aplicación en los ambientes preproductivos (desarrollo y certificación) y de igual modo en producción. Dentro de este *pipeline*, se ejecutan pruebas de aceptación, performance y seguridad que deben ejecutarse correctamente en cada ambiente. También, dentro del *pipeline* se tienen tareas de *rollback*, que, de ser necesario, permitiera regresar al último despliegue exitoso en caso de que ocurra algún inconveniente durante el paso a producción.

Finalmente, viene el paso a producción, con esto, los cambios introducidos quedan disponibles en la versión del aplicativo que utilizan los usuarios; para este proceso, se requiere de la creación de una orden de cambio que debe ser aprobada por el líder del equipo y asociada a esta, se crea también un *Definition of done* (DOD) mediante el cual el PO confirma que el software está listo para su paso a producción. La Figura 1 ilustra, de manera resumida, la arquitectura seguida en el desarrollo de este proyecto.

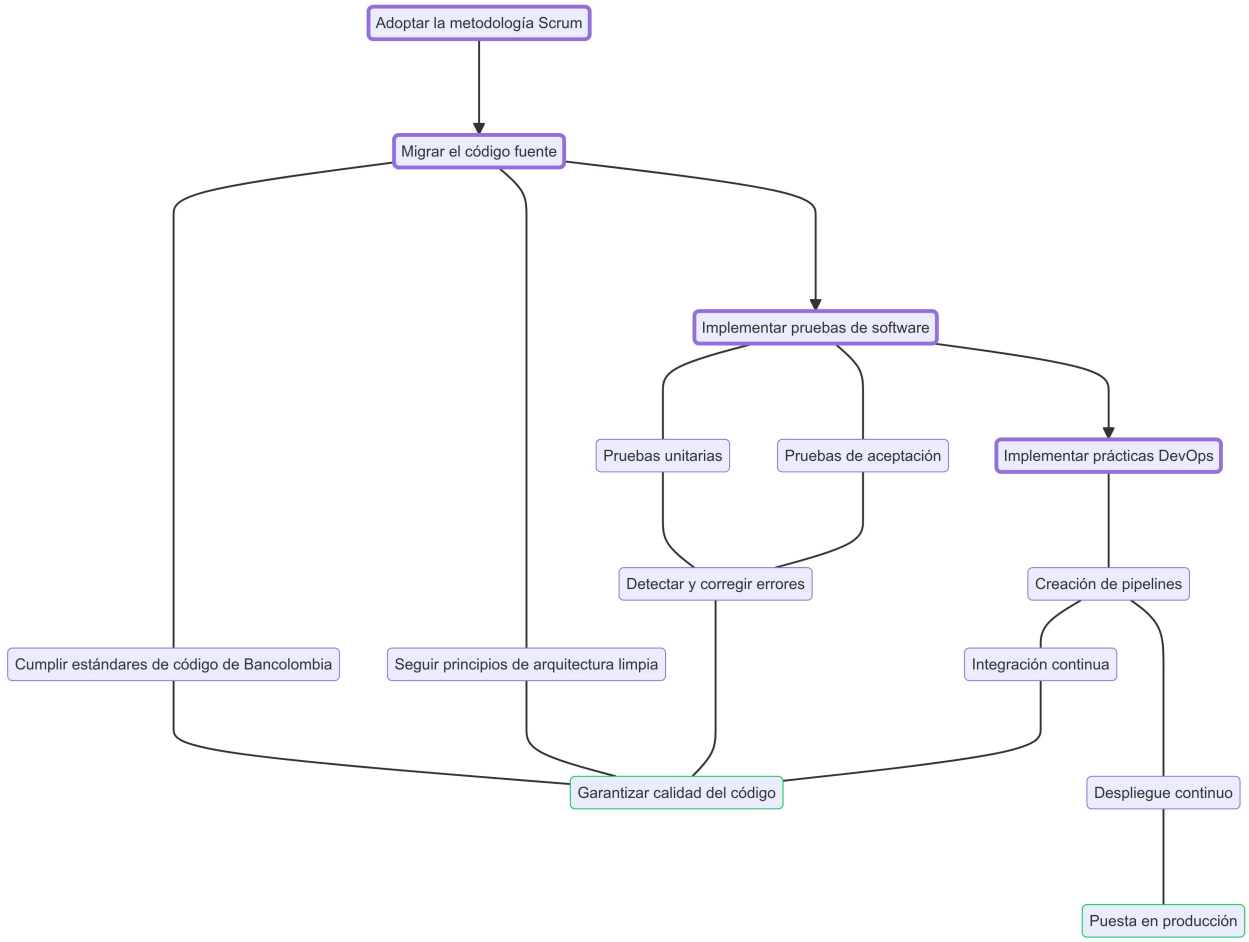

Fig. 1. Diagrama resumen de la metodología

V. RESULTADOS Y ANÁLISIS

El trabajo realizado, permitió concretar la migración a arquitectura limpia de los microservicios listados a continuación:

- Simulador Crediágil
- Simulador crédito de libre inversión
- Simulador crédito de vivienda
- Simulador libranza empleados

Estos simuladores permiten a los clientes de Bancolombia obtener una estimación detallada de las características de algunos productos crediticios ofrecidos por el banco. Por ejemplo, proveen información aproximada sobre las cuotas y las distintas opciones de pago, basadas en los datos ingresados por el cliente, tales como edad, monto del crédito, número de cuotas, entre otros.

Para cada uno de estos simuladores se generó el respectivo repositorio con el código fuente estructurado bajo arquitectura limpia, las pruebas unitarias, las pruebas de aceptación y de rendimiento, También se crearon los repositorios en *SonarQube* y los pipelines de CI y CD. La adopción de arquitecturas limpias para estos simuladores supone numerosas ventajas, entre las cuales destacan la clara separación de responsabilidades y el bajo acoplamiento entre los diferentes componentes del software, disminuyendo así la complejidad del código, además se constituye un escenario que facilita futuros mantenimientos o modificaciones.

Por su parte, una correcta implementación de las practicas *DevOps* blinda el proceso de desarrollo y despliegue de errores en el código o la configuración a la vez que agiliza dicho proceso y mejora la coordinación entre los distintos roles implicados. Adicional al cambio de arquitectura, la migración de los simuladores ha permitido un avance en diversas métricas relacionadas con la calidad del código, cuya evaluación se ha llevado a cabo mediante *SonarQube*, esta herramienta al realizar una serie de comprobaciones automatizadas sobre el código fuente, facilita la generación de informes con diferentes métricas y la identificación de desviaciones en la calidad del código.

Entre estas métricas destacan la cantidad de horas de deuda técnica, la cantidad de "*code smells*", la cobertura de pruebas unitarias, y la complejidad ciclomática y cognitiva del código.

La Tabla 1 presenta los resultados obtenidos a través de *SonarQube* y ofrece una comparativa de los resultados obtenidos en las versiones previas de los simuladores, bajo la arquitectura MVC y su contraparte en arquitectura limpia.

TABLA I
COMPARATIVO DE LAS MÉTRICAS DE CALIDAD DE CÓDIGO

Simulador	**Medida**	**Antes**	**Ahora**
CrediÁgil	Deuda técnica	3.167 h	0.67 h
	Code Smells	10	2
	Cobertura de pruebas unitarias	79.2%	87.0%
	Complejidad ciclomática	150	72
	Complejidad cognitiva	36	32
Crédito de libre inversión	Deuda técnica	6.05 h	2 h
	Code Smells	36	6
	Cobertura de pruebas unitarias	81.0%	93.2%
	Complejidad ciclomática	218	111
	Complejidad cognitiva	71	64
Crédito de vivienda	Deuda técnica	6.35 h	2.33 h
	Code Smells	50	7
	Cobertura de pruebas unitarias	77.6%	78.8%
	Complejidad ciclomática	488	318
	Complejidad cognitiva	161	155
Libranza	Deuda técnica	7.13 h	1.92h
	Code Smells	45	6
	Cobertura de pruebas unitarias	73.8%	78.3%
	Complejidad ciclomática	229	97
	Complejidad cognitiva	56	40

El análisis de la comparativa entre el estado previo y posterior de los simuladores que fueron objeto de la migración ofrece una visión clara de las mejoras alcanzadas en términos de calidad del código, cobertura de pruebas unitarias y complejidad del código. Demostrando el impacto positivo de las migraciones realizadas sobre los simuladores del portal de contenidos de Bancolombia.

En la Figura 2 se detallan los avances en términos de deuda técnica, mostrando el número de horas reportadas por *SonarQube*, este es un estimado del tiempo requerido para solventar las deficiencias encontradas en el código.

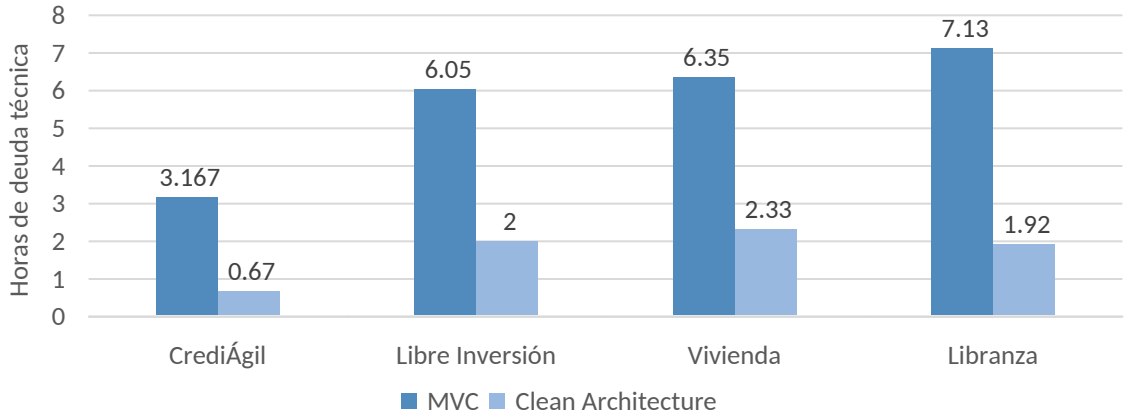

Fig. 2. Gráfico comparativo de horas de deuda técnica

Tras las migraciones en los 4 simuladores, las horas de deuda técnica se redujeron en un 70%. Adicionalmente, como se puede observar en la Figura 3 el número de *code smells* disminuyó de manera considerable, lo que indica una amplia mejora en la calidad del código.

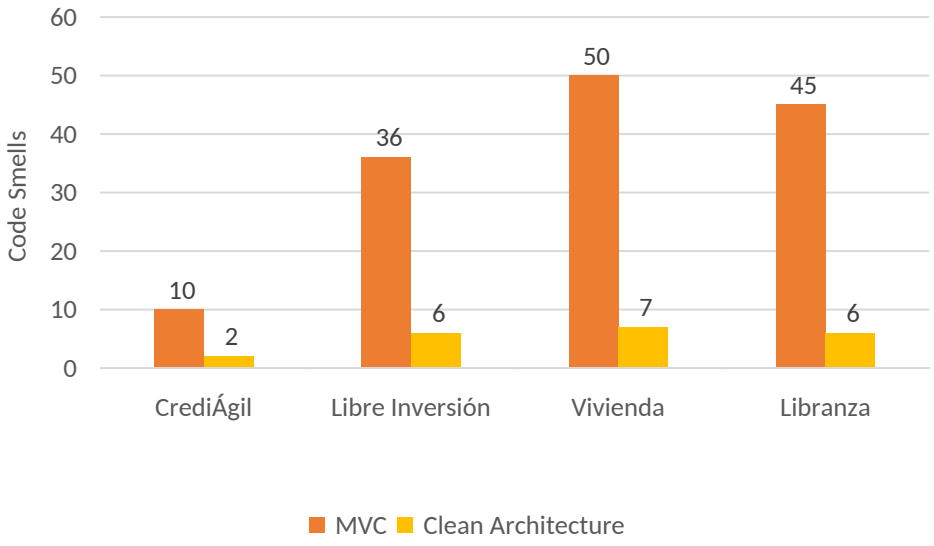

Fig. 3. Gráfico comparativo de cantidad de *code smells*

De manera similar, en las Figuras 4 y 5, se puede encontrar que también hubo mejoras en cuanto a la complejidad ciclomática y cognitiva del código, indicando que el código es más fácil de entender y modificar para los desarrolladores [20], lo que en última instancia reduce el esfuerzo de codificación y el riesgo de tener código con errores [20], [21] y evidencia el impacto positivo de la implementación de arquitectura limpia para reducir la complejidad del código [7] al desacoplar la lógica y simplificar las dependencias.

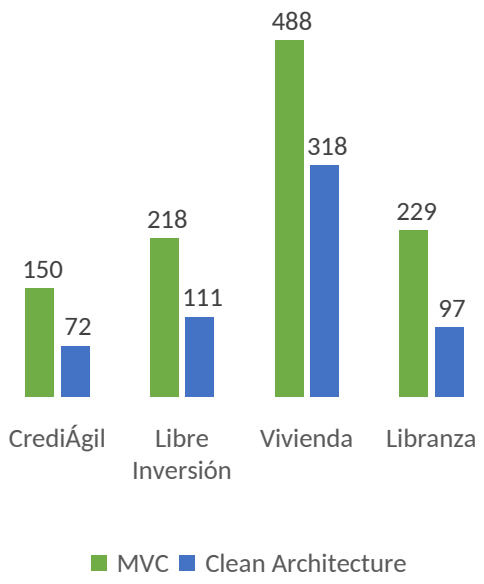

Fig. 4. Gráfico comparativo de complejidad ciclomática

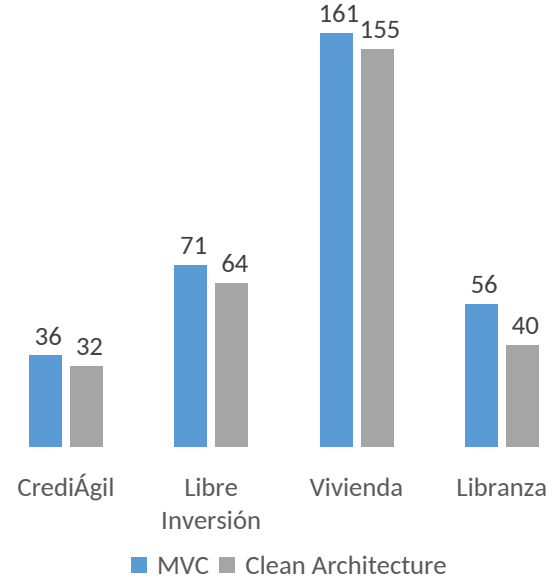

Fig. 5. Gráfico comparativo de complejidad cognitiva

Los avances en estos 4 indicadores redundan en una base de código más mantenible, lo que facilita futuras modificaciones y reduce el riesgo de introducir errores y fallos. Además, se disminuyen los costos a largo plazo asociados con la corrección de problemas acumulados y permite a los equipos de desarrollo centrarse en nuevas funcionalidades en lugar de reparaciones. En conjunto, esto conduce a una mayor eficiencia operativa y un tiempo de entrega más rápido de futuras características.

Aunado a lo anterior, las migraciones también permitieron tener avances en la cobertura de pruebas unitarias, como se puede ver en la Figura 6.

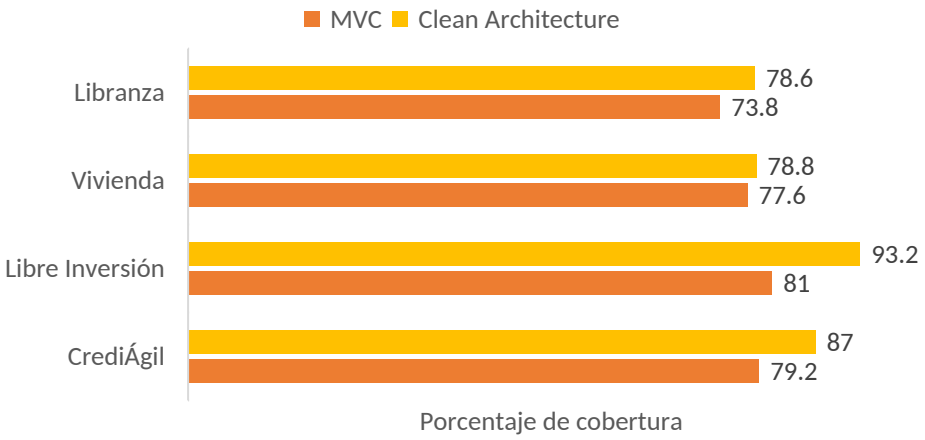

Fig. 6. Gráfico comparativo de cobertura de pruebas unitarias

A pesar de que por lineamiento del banco la cobertura para todos los desarrollos debe ser superior al 70% y que las versiones previas ya tenían coberturas superiores a este límite, se logró aumentar esta cobertura en un 10%.

Este avance mejora la confiabilidad del código al asegurar que una mayor parte del código esté verificado contra errores antes de su despliegue. Al detectar errores en etapas tempranas, se reducen los costos y el tiempo de corrección, facilitando un desarrollo más ágil y eficiente. Además, las pruebas unitarias mejoran la mantenibilidad del código al proporcionar una base sólida para futuras modificaciones y refactorizaciones, asegurando que los cambios no introduzcan nuevos errores. Así, incrementa la confianza en el software, acelera el ciclo de desarrollo y ayuda a entregar productos más robustos y estables.

VI. CONCLUSIONES

La adopción de arquitectura limpia permitió estructurar el código de manera modular y desacoplada, facilitando así la identificación y reducción de la deuda técnica existente. Esta restructuración no solo mejoró la calidad del código, sino que también simplifica su mantenimiento y evolución futura.

La implementación de prácticas DevOps, como la integración y el despliegue continuos, junto con la automatización de pruebas, resultó en un ciclo de desarrollo más rápido y confiable, permitiendo detectar errores en etapas tempranas del ciclo de desarrollo, reduciendo costos y tiempos de corrección, y mejorando la calidad final del producto.

La combinación de una arquitectura bien definida y la adopción de prácticas DevOps ha llevado a un aumento en la cobertura de pruebas unitarias y a una mayor robustez del código. Esto ha incrementado la confianza en el software desplegado, asegurando que los cambios realizados no introduzcan nuevos errores y que el producto final cumpla con los estándares de calidad esperados.

REFERENCIAS

[1] Bancolombia, «Wiki Visepresidencia de servicios de tecnología».

[2] M. Fowler, «Technical Debt», *martinfowler.com*. mayo de 2019. [En línea]. Disponible en: https://martinfowler.com/bliki/TechnicalDebt.html

[3] T. Asana, «Qué es la deuda técnica y cómo saldarla (con ejemplos)», *Asana.com*. enero de 2024. [En línea]. Disponible en: https://asana.com/es/resources/technical-debt

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, y C. Seaman, «Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)», *Dagstuhl Rep.*, vol. 6, n.º 4, 2016.

[5] P. V. Isha y H. V. Prabhu, «An Approach to Clean Architecture for Microservices Using Python», en *7th IEEE International Conference on Computational Systems and Information Technology for Sustainable Solutions, CSITSS 2023 - Proceedings*, 2023. doi: 10.1109/CSITSS60515.2023.10334229.

[6] D. Sas y P. Avgeriou, «An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells», *IEEE Trans. Softw. Eng.*, vol. 49, n.º 8, 2023, doi: 10.1109/TSE.2023.3286179.

[7] Y. N. Nugroho, D. S. Kusumo, y M. J. Alibasa, «Clean Architecture Implementation Impacts on Maintainability Aspect for Backend System Code Base», en *2022 10th International Conference on Information and Communication Technology, ICoICT 2022*, 2022. doi: 10.1109/ICoICT55009.2022.9914890.

[8] A. Chidi, «Clean Architecture: The Key to Developing High-Quality Software», *Qalbit.com*. febrero de 2023.

[9] R. C. Martin, «The Clean Architecture», *The Clean Code Blog*. agosto de 2013. [En línea]. Disponible en: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[10] A. M. Gómez, «Clean Architecture — Aislando los detalles», *Medium.com*. marzo de 2020. [En línea]. Disponible en: https://medium.com/bancolombia-tech/clean-architecture-aislando-los-detalles-4f9530f35d7a

[11] M. Alawneh y I. M. Abbadi, «Expanding DevOps Principles and Best Practices Based on Practical View», en *Proceedings - 2022 23rd International Arab Conference on Information Technology, ACIT 2022*, 2022. doi: 10.1109/ACIT57182.2022.9994216.

[12] M. F. D. Acosta y G. A. G. Mireles, «Identifying Activities for Enhancing Software Quality in DevOps Settings», en *2021 10th International Conference On Software Process Improvement (CIMPS)*, IEEE, oct. 2021, pp. 84-89. doi: 10.1109/CIMPS54606.2021.9652761.

[13] T. Rangnau, R. V. Buijtenen, F. Fransen, y F. Turkmen, «Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines», en *Proceedings - 2020 IEEE 24th International Enterprise Distributed Object Computing Conference, EDOC 2020*, 2020. doi: 10.1109/EDOC49727.2020.00026.

[14] M. Nabil, «Elevating Code Quality: The Power of Static Code Analysis in Modern Software Development», *Medium.com*. octubre de 2023. [En línea]. Disponible en: https://medium.com/aviv-product-tech-blog/elevating-code-quality-the-power-of-static-code-analysis-in-modern-software-development-e0316e303afb

[15] T. Offerman, R. Blinde, C. J. Stettina, y J. Visser, «A Study of Adoption and Effects of DevOps Practices», en *2022 IEEE 28th International Conference on Engineering, Technology and Innovation, ICE/ITMC 2022 and 31st International Association for Management of Technology, IAMOT 2022 Joint Conference - Proceedings*, 2022. doi: 10.1109/ICE/ITMC-IAMOT55089.2022.10033313.

[16] J. Sutherland, *Scrum Handbook*. Accedido: 13 de julio de 2024. [En línea]. Disponible en: https://scrummaster.dk/lib/AgileLeanLibrary/People/JeffSutherland/scrumhandbook.pdf

[17] K. Schwaber y J. Sutherland, «The Scrum Guide». 2012. [En línea]. Disponible en: https://api.semanticscholar.org/CorpusID:114128971

[18] B. Haugset y G. K. Hanssen, «Automated Acceptance Testing: A Literature Review and an Industrial Case Study», en *Agile 2008 Conference*, ago. 2008, pp. 27-38. doi: 10.1109/Agile.2008.82.

[19] N. Nasir, «Acceptance Testing in Agile Software Development».

[20] V. Lenarduzzi, T. Kilamo, y A. Janes, «Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception». arXiv, 14 de marzo de 2023. doi: 10.48550/arXiv.2303.07722.

[21] «Automatizing Software Cognitive Complexity Reduction | IEEE Journals & Magazine | IEEE Xplore». Accedido: 1 de julio de 2024. [En línea]. Disponible en: https://ieeexplore-ieee-org.udea.lookproxy.com/document/9686676