UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA

ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS

TESIS

METODOLOGÍA DE OPTIMIZACIÓN EN CI/CD PARA LA COMPILACIÓN Y DESPLIEGUE DE APLICACIONES MODULARES EN ENTORNOS EMPRESARIALES

PRESENTADO POR:

Br. Edwar Yuri Cassa Lipa

PARA OPTAR AL TÍTULO PROFESIONAL DE:

Ingeniero Informático y de Sistemas

ASESOR:

Dr. Rony Villafuerte Serna

Cusco - Perú
2025

Universidad Nacional de San Antonio Abad del Cusco

INFORME DE SIMILITUD

(Aprobado por Resolución Nro.CU-321-2025-UNSAAC)

El que suscribe, el **Asesor** DR. RONY VILLAFUERTE SERNA ..... quien aplica el software de detección de similitud al trabajo de investigación/tesis titulada: METODOLOGÍA DE OPTIMIZACIÓN EN CI/CD PARA LA COMPILACIÓN Y DESPLIEGUE DE APLICACIONES MODULARES EN ENTORNOS EMPRESARIALES .....

Presentado por: EDWAR YURI CASSA LIPA ..... DNI N° 70401739 ;

presentado por: ..... DNI N°: .....

Para optar el título Profesional/Grado Académico de INGENIERO INFORMÁTICO Y DE SISTEMAS .....

Informo que el trabajo de investigación ha sido sometido a revisión por 3 veces, mediante el Software de Similitud, conforme al Art. 6° del ***Reglamento para Uso del Sistema Detección de Similitud en la UNSAAC*** y de la evaluación de originalidad se tiene un porcentaje de 2 %.

Evaluación y acciones del reporte de coincidencia para trabajos de investigación conducentes a grado académico o título profesional, tesis

Porcentaje	Evaluación y Acciones	Marque con una (X)
Del 1 al 10%	No sobrepasa el porcentaje aceptado de similitud.	X
Del 11 al 30 %	Devolver al usuario para las subsanaciones.	
Mayor a 31%	El responsable de la revisión del documento emite un informe al inmediato jerárquico, conforme al reglamento, quien a su vez eleva el informe al Vicerrectorado de Investigación para que tome las acciones correspondientes; Sin perjuicio de las sanciones administrativas que correspondan de acuerdo a Ley.	

Por tanto, en mi condición de Asesor, firmo el presente informe en señal de conformidad y **adjunto** las primeras páginas del reporte del Sistema de Detección de Similitud.

Cusco, 7 de ABRIL de 20 26

.....
Firma

Post firma DR. RONY VILLAFUERTE SERNA

Nro. de DNI 23957778

ORCID del Asesor 0000-0003-4607-522X

Se adjunta:

1. Reporte generado por el Sistema Antiplagio.
2. Enlace del Reporte Generado por el Sistema de Detección de Similitud: **oid:** 27259:575431004

Edwar Yuri Cassa Lipa

OptimizacionERP_v3.pdf

Universidad Nacional San Antonio Abad del Cusco

Detalles del documento

Identificador de la entrega
trn:oid:::27259:575431004

Fecha de entrega
6 abr 2026, 5:20 p.m. GMT-5

Fecha de descarga
6 abr 2026, 5:28 p.m. GMT-5

Nombre del archivo
ERP.pdf

Tamaño del archivo
5.8 MB

91 páginas

18.699 palabras

107.028 caracteres

2% Similitud general

El total combinado de todas las coincidencias, incluidas las fuentes superpuestas, para ca...

Filtrado desde el informe

- Bibliografía
- Texto citado
- Coincidencias menores (menos de 20 palabras)

Exclusiones

- N.º de coincidencias excluidas

Fuentes principales

2% Fuentes de Internet
0% Publicaciones
1% Trabajos entregados (trabajos del estudiante)

Marcas de integridad

N.º de alertas de integridad para revisión

No se han detectado manipulaciones de texto sospechosas.

Los algoritmos de nuestro sistema analizan un documento en profundidad para buscar inconsistencias que permitirían distinguirlo de una entrega normal. Si advertimos algo extraño, lo marcamos como una alerta para que pueda revisarlo.

Una marca de alerta no es necesariamente un indicador de problemas. Sin embargo, recomendamos que preste atención y la revise.

Dedicatoria

A mis padres y hermanos, por su inmenso amor;
ustedes significan mucho para mi.

Agradecimientos

Quiero expresar mi profundo agradecimiento a aquellas personas que han hecho posible la realización de este proyecto.

A mis amigos por su apoyo incondicional y por sacarme siempre una sonrisa. Gracias por estar a mi lado.

A mi asesor, por su tiempo, orientación y esfuerzo en la corrección de este documento. gracias por su paciencia y dedicación durante este proceso.

Y a todas las personas que han formado parte de mi camino, les estoy profundamente agradecido por su contribución y por hacer que esta aventura valga la pena.

Resumen

Actualmente, muchas empresas están migrando de arquitecturas monolíticas a modelos modulares para mejorar la escalabilidad y el mantenimiento de sus sistemas. Sin embargo, esta transición plantea nuevos retos en los procesos de compilación y despliegue continuo (CI/CD), los cuales pueden volverse ineficientes si no se aplican estrategias adecuadas. Esta tesis propone un modelo optimizado de CI/CD para aplicaciones empresariales modulares sin depender completamente del enfoque de micro-frontends, utilizando subaplicaciones independientes y librerías compartidas como parte de una arquitectura modular basada en Angular, compilación selectiva basada en cambios, y despliegues independientes orquestados con Docker, Helm y Kubernetes. El objetivo principal es reducir tiempos de compilación, evitar tareas redundantes y mejorar la eficiencia general del pipeline. El caso de estudio fue un sistema ERP real de la empresa "Smart Cities Perú", donde se logró reducir el tiempo de compilación de mas de un 60 %, con beneficios adicionales en productividad de desarrollo y costos operativos simulados en entornos de nube. Más allá del resultado aplicado, el aporte académico de esta investigación radica en la formulación de un enfoque replicable y adaptable a otros sistemas empresariales con características similares. De este modo, la propuesta no solo evidencia beneficios prácticos en productividad y costos operativos, sino que también contribuye a la literatura académica en torno a la optimización de procesos CI/CD en arquitecturas modulares.

Palabras clave: CI/CD, Docker, Kubernetes, ERP.

Abstract

Currently, many companies are migrating from monolithic architectures to modular models to improve the scalability and maintainability of their systems. However, this transition introduces new challenges in continuous integration and continuous deployment (CI/CD) processes, which can become inefficient if appropriate strategies are not applied. This thesis proposes an optimized CI/CD model for modular enterprise applications without fully relying on the micro-frontend approach. The model leverages independent sub-applications and shared libraries as part of a modular Angular-based architecture, selective compilation based on code changes, and independent deployments orchestrated with Docker, Helm, and Kubernetes. The main objective is to reduce compilation times, avoid redundant tasks, and improve overall pipeline efficiency. The case study was conducted on a real ERP system from the company "Smart Cities Perú", where compilation time was reduced by more than 60 %, with additional benefits in developer productivity and simulated operational cost savings in cloud environments. Beyond the applied result, the academic contribution of this research lies in the formulation of a replicable and adaptable approach for other enterprise systems with similar characteristics. Thus, the proposal not only demonstrates practical benefits in terms of productivity and operational efficiency but also contributes to the academic literature on CI/CD process optimization in modular architectures.

Keywords: CI/CD, Docker, Kubernetes, ERP.

Introducción

En los últimos años, el desarrollo de software empresarial ha experimentado una transición progresiva desde arquitecturas monolíticas hacia modelos más flexibles y escalables, como las arquitecturas modulares, basadas en microservicios o en la separación de funcionalidades en subaplicaciones. Esta evolución busca mejorar el mantenimiento, la escalabilidad y la velocidad de entrega del software, especialmente en sistemas complejos como los ERP (Enterprise Resource Planning), que abarcan múltiples módulos funcionales.

Sin embargo, a diferencia de otros tipos de aplicaciones (como paginas web o aplicaciones móviles), los ERP empresariales continúan en muchos casos bajo arquitecturas monolíticas. Incluso cuando se avanza hacia modelos modulares, la literatura académica y la práctica profesional muestran un vacío: no existen suficientes estudios comparativos ni guías metodológicas claras sobre cómo adaptar y optimizar los pipelines de integración y despliegue continuo (CI/CD) en este tipo de sistemas. Esta carencia dificulta que las organizaciones cuenten con referentes para modularizar sus procesos de compilación y despliegue de manera eficiente.

En este contexto, los retos técnicos se hacen evidentes. La presencia de múltiples módulos puede provocar que pequeños cambios generen la necesidad de compilar componentes que no fueron modificados, afectando los tiempos de respuesta tanto del equipo de desarrollo y en el proceso de despliegue, aumentando el uso de recursos computacionales. La falta de estrategias específicas para ERP incrementa la relevancia de este problema, pues se trata de sistemas críticos en los que la disponibilidad y agilidad son fundamentales.

La eficiencia del pipeline CI/CD se vuelve crítica en contextos donde el tiempo es un factor determinante, como en sistemas en producción que requieren cambios rápidos ante errores o actualizaciones. Optimizar el flujo de entrega continua no solo mejora la productividad del equipo de desarrollo, sino que también permite reducir los costos operativos y aumentar la confiabilidad del proceso de despliegue. Además, este trabajo aporta un valor académico adicional al proponer un modelo metodológico específico y replicable para ERP modulares, cubriendo un vacío en la investigación sobre la optimización de pipelines en este tipo de sistemas.

El problema central abordado en esta tesis es la ineficiencia en los procesos de compilación y despliegue de aplicaciones empresariales modulares, ocasionada por pipelines que no distinguen los cambios entre módulos y requieren compilar la aplicación completa incluso ante modificaciones mínimas. A ello se suma la ausencia de estudios comparativos y metodológicos que orienten la implementación de pipelines optimizados en ERP, lo cual limita la capacidad de mejorar de forma sistemática su rendimiento operativo.

Dado este escenario, resulta fundamental proponer un modelo de CI/CD que se adapte a arquitecturas modulares, permitiendo compilar , construir y desplegar únicamente los módulos afectados por los cambios, integrando además herramientas modernas de orquestación y contenedorización.

El objetivo general de esta tesis es implementar y validar un modelo metodológico optimizado en CI/CD aplicable a sistemas ERP empresariales, que mejore la eficiencia de los procesos de compilación y despliegue y que al mismo tiempo pueda servir como guía metodológica replicable para entornos similares. Para lograrlo, se plantean los siguientes objetivos específicos:

- Reducir los tiempos de compilación mediante la adopción de una arquitectura modular que permita la compilación y despliegue independientes por módulo.
- Implementar pipelines CI/CD eficientes que utilicen compilación selectiva, versionado de imágenes Docker y despliegues separados.
- Optimizar el uso de recursos computacionales y reducir los costos operativos asociados a la infraestructura CI/CD.

La propuesta se desarrolló sobre un caso práctico real: un sistema ERP perteneciente a la empresa "Smart Cities Perú". A partir de una arquitectura inicialmente monolítica, se llevó a cabo un proceso de modularización utilizando subaplicaciones y librerías en Angular. Se diseñó un script de detección de cambios para compilar únicamente los módulos modificados, y se configuró un pipeline CI/CD en GitLab que automatiza la construcción de imágenes Docker y su despliegue en Kubernetes mediante Helm respecto a estos cambios aplicados.

El análisis de resultados se realizó comparando los tiempos totales de ejecución de los pipelines antes y después de la optimización, junto con una simulación en ahorro de costos operativos en la nube, usando como referencia instancias bajo demanda de AWS.

La presente tesis se estructura de la siguiente manera:

- **Capítulo 1:** Introducción, donde se presenta el contexto del problema, los objetivos de la investigación, la justificación del estudio y la metodología general aplicada.
- **Capítulo 2:** Marco teórico, en el que se abordan los conceptos fundamentales relacionados con CI/CD, arquitecturas modulares en Angular, uso de contenedores con Docker, orquestación con Kubernetes, entre otros temas relevantes para el desarrollo del proyecto.
- **Capítulo 3:** Análisis del estado actual del sistema ERP como caso de estudio, seguido del diseño de la propuesta, donde se describe la arquitectura modular adoptada, las configuraciones del pipeline CI/CD (antes y después de la optimización), los scripts implementados, así como los componentes desarrollados y las mejoras aplicadas.
- **Capítulo 4:** Evaluación del rendimiento del nuevo pipeline CI/CD, incluyendo las métricas obtenidas durante las fases de compilación, construcción de imágenes y despliegue. Se presentan estadísticas comparativas y datos obtenidos directamente del entorno de ejecución.

- **Capítulo 5:** Análisis y discusión de los resultados, tanto técnicos como operativos, junto con una simulación de ahorro de costos en infraestructura, basada en escenarios de ejecución en servicios de nube pública como AWS.
- **Capítulo 6:** Conclusiones generales del trabajo, acompañadas de recomendaciones para futuras investigaciones, mejoras técnicas o posibles ampliaciones del modelo propuesto.

Glosario

AWS Amazon Web Services..

Backend Parte de una aplicación que maneja la lógica de negocio y el procesamiento de datos.

Chart Paquete que facilita el despliegue y gestión de aplicaciones en Kubernetes..

docker Herramienta que permite empaquetar aplicaciones y sus dependencias en unidades llamadas contenedores..

ERP Sistema de planificación de recursos empresariales.

Frontend Parte de una aplicación que interactura con el usuario.

ITIL Marco de buenas prácticas para la gestión de servicios de tecnologías de la información..

Runner Agente liviano y escalable que ejecuta el trabajo de CI descrito en el archivo de configuración .yaml.

Stages Pasos que se realizan dentro de un pipeline..

Tag Etiqueta asociada a un commit generalmente para hacer seguimiento a versiones de una aplicación..

Índice general

Índice de figuras

Índice de Tablas

Capítulo 1

Aspectos generales

1. Planteamiento del problema

1.1. Descripción del problema

Actualmente, muchas empresas se preocupan por la eficiencia en el desarrollo de sus aplicaciones, las cuales son esenciales para satisfacer las necesidades de sus clientes. Esta investigación aborda el desafió de cómo mejorar los tiempos de compilación y despliegue de aplicaciones empresariales. Muchas empresas medianas y grandes aún teniendo sus aplicaciones monolíticas, caracterizadas por agrupar con una estructura centralizada en el directorio de código, presentan problemas de escalabilidad y mantenimiento a medida que crecen. Un cambio en cualquier parte del código implica la recompilación y despliegue de toda la aplicación, lo que resulta en tiempos de espera largos y procesos ineficientes, especialmente si la aplicación es grande. Esta falta de agilidad en el ciclo de desarrollo no solo retrasa la capacidad de respuesta ante cambios requeridos, sino que también genera una presión en el equipo de desarrollo limitando su productividad y afectando la satisfacción del cliente.

La transición a una arquitectura modular y micro-frontend promete solucionar estos problemas, al permitir descomponer la aplicación en subaplicaciones. Esta estructura modular facilita el desarrollo y compilación de manera independiente, lo que, en teoría, debería reducir significativamente los tiempos de compilación y mejorar la eficiencia general del proceso. Sin embargo, este proceso trae consigo nuevos desafíos. Las empresas que adoptan este enfoque deben lidiar con la complejidad técnica de descomponer la aplicación monolítica y ajustar sus procesos de compilación y despliegue para garantizar que los resultados esperados se realicen correctamente sin afectar la estabilidad del sistema.

La propuesta que se mencionará, hará uso de estrategias de CI/CD adecuada para gestionar la integración y entrega continua de estas subaplicaciones con el fin de mejorar los procesos de desarrollo y entrega al cliente.

1.2. Identificación del problema

Al desarrollar aplicaciones empresariales de gran tamaño, como los sistemas ERP, uno de los mayores desafíos es optimizar los tiempos de compilación e implementación. A medida que aumenta la complejidad de estas aplicaciones, aumentan los tiempos de entrega durante los ciclos de desarrollo, lo que afecta la capacidad de las empresas para responder rápidamente a los cambios del mercado o a las nuevas necesidades del cliente.

Un ejemplo de este problema se observa en la empresa "Smart Cities Perú", cuyo sistema ERP ha alcanzado un tamaño considerable al contar con 32 módulos funcionales, lo que lo clasifica como un sistema de gran tamaño según el marco teórico, el cual provoca demoras en los procesos de compilación y despliegue del mismo. Esta aplicación requiere que cada vez que se realice un cambio en cualquier parte del sistema, se recompile y despliegue todo el proyecto, lo que ralentiza el ciclo de desarrollo y limita la agilidad de la empresa.

Este estudio se enfoca en aplicar una metodología propuesta de optimización de los tiempos de compilación y despliegue en aplicaciones empresariales, utilizando un ERP como un caso práctico para ilustrar cómo sistemas grandes y complejos pueden beneficiarse de un enfoque modular. La modularización, junto con tecnologías como Docker y Kubernetes, permite gestionar cada módulo de forma independiente, mejorando la eficiencia del desarrollo asegurando que los cambios se integren de manera rápida sin comprometer la estabilidad de la aplicación.

Adicionalmente, para validar el impacto del modelo metodológico propuesto, se recopilaron datos reales de tiempo de ejecución de los pipelines CI/CD desde la infraestructura de la empresa antes mencionada, institución que utiliza este ERP en su entorno de producción. Dichos datos permitieron medir de forma objetiva los resultados de la optimización propuesta, sin comprometer información sensible.

2. Formulación del problema

2.1. Problema general

Ineficiencia de los procesos de compilación y despliegue en aplicaciones empresariales, principalmente debido a la falta de prácticas de CI/CD adaptadas a los sistemas, afectando la productividad, estabilidad y rendimiento del software.

2.2. Problemas específicos

- La necesidad de recompilar toda la aplicación ante cualquier cambio, incluso si afecta solo a una parte del sistema, aumenta significativamente los tiempos de compilación y despliegue.
- La arquitectura monolítica limita la capacidad de aplicar pipelines CI/CD eficientes para la compilación y despliegue automático para módulos independientes.
- El consumo excesivo de recursos computacionales durante los procesos de compilación y despliegue en entornos monolíticos incrementa los costos operativos.

3. Objetivos

3.1. Objetivo general

Implementar un modelo metodológico optimizado de CI/CD para mejorar la eficiencia de los procesos de compilación y despliegue en aplicaciones ERP empresariales, adaptado a estos sistemas, con el fin de incrementar la productividad y la calidad del software.

3.2. Objetivos específicos

- Reducir los tiempos de compilación implementando una arquitectura modular que permita compilaciones selectivas y desplegar de manera independiente cada módulo de la aplicación.
- Implementar pipelines CI/CD eficientes adaptado a la arquitectura modular, que incluyan versionado de imágenes Docker y soporten la compilación y despliegue independiente de módulos.
- Optimizar y reducir costos operativos configurando pipelines CI/CD que minimicen las tareas redundantes y prioricen el uso eficiente de recursos en entornos de desarrollo y producción.

4. Justificación

4.1. Conveniencia

Si bien las empresas se destacan por la eficiencia de sus procesos, esta investigación promete una agilidad en el desarrollo, así el equipo podrá implementar mejoras y resolver problemas con mayor rapidez lo cual facilita la entrega continua, esto es importante en aplicaciones de gran tamaño como un ERP que requieren constantes cambios para adaptarse a las necesidades comerciales, a su vez buscar la escalabilidad del sistema el cual es clave para empresas en crecimiento.

Por ello, es muy importante aplicar nuevas técnicas para contrarrestar estos aspectos garantizando siempre su uso correcto y la estabilidad del sistema. Entre las mejoras, estos incluyen pipelines en CI/CD, construcción de imágenes versionadas, este último para organizar de manera óptima los módulos contenerizados en imágenes desplegadas en entornos de desarrollo y producción.

4.2. Relevancia

La investigación aborda el impacto en la competitividad empresarial ya que al implementar procesos en CI/CD podrán responder mejor a las demandas del mercado, lo que tendrá un impacto directo en mejorar la capacidad para desarrollar y desplegar software de manera más ágil y eficiente. Además al aplicar pipelines, Docker, Helm y Kubernetes, no solo se contribuye al conocimiento en CI/CD, estableciendo mejores practicas y referencias para otros desarrolladores de enfrentan problemas similares en aplicaciones monolíticas grandes, sino que también impulsa a la innovación en la forma de administrar y desplegar aplicaciones a gran escala.

4.3. Implicancias prácticas

La implementación de un proceso de CI/CD optimizado para aplicaciones de gran escala proporciona un enfoque práctico en la empresas, estos incluyen:

- Reducción de tiempos de desarrollo y despliegue el cual permitirá responder a cambios solicitados en el sistema.
- Facilitar la gestión y actualización de grandes aplicaciones, esto facilitará realizar un mantenimiento mas escalable.
- Mejorar la eficiencia del desarrollo compilando solo los módulos que sufrieron cambios a nivel de código en lugar del sistema completo.

4.4. Valor teórico

Con este trabajo se busca contribuir al conocimiento en los campos de CI/CD y arquitectura de software, mostrando cómo tecnologías como Docker, Kubernetes y Helm pueden integrarse en pipelines de CI/CD para optimizar procesos en aplicaciones de gran escala. Al mismo tiempo, se aporta un valor académico al proponer un modelo metodológico concreto y replicable para ERP modulares, lo que permite cubrir un vacío en la literatura respecto a la optimización de pipelines en este tipo de sistemas.

4.5. Utilidad metodológica

El estudio proporciona un modelo práctico para empresas que deseen optimizar sus procesos usando pipelines de CI/CD y aplicarlos a sistemas similares. Estos incluyen:

- Proveer un enfoque detallado para modularizar aplicaciones monolitícas mediante la separación tanto en librerías y en subaplicaciones.
- Adaptar el enfoque y tecnologías propuestas por otros profesionales o equipos de desarrollo.
- Realizar métricas de evaluación para medir la eficiencia de los procesos CI/CD en otras aplicaciones.

5. Delimitación de estudio

5.1. Delimitación espacial

Esta investigación se centrará en optimizar los procesos de CI/CD del un ERP perteneciente a la empresa "Smart Cities Perú". Actualmente, este sistema emplea tecnologías de contenedorización y despliegue, aunque de forma ineficiente en la compilación y despliegue de la aplicación completa, lo cual limita la agilidad en la entrega de nuevas funcionalidades.

La validación de resultados, específicamente en cuanto a tiempos de ejecución y comportamiento real del pipeline CI/CD, se realizó mediante el análisis de datos técnicos extraídos del sistema GitLab local de la empresa Smart Cities Perú, institución donde tiene desplegado dicho ERP.

De este modo, la investigación se desarrolló en un contexto empresarial real con soporte técnico en ambos frentes: el diseño y desarrollo del modelo y la validación empírica a partir de métricas registradas en el entorno de uso de Smart Cities Perú. Esta combinación permite obtener resultados reales, sin comprometer datos sensibles de la empresa.

5.2. Delimitación temporal

El desarrollo de esta investigación se llevó a cabo durante un período de siete meses, desde julio del 2024 hasta febrero de 2025, tiempo durante el cual se implementó, probó y evaluó el nuevo pipeline de CI/CD para observar su impacto en la eficiencia del despliegue del ERP.

6. Método

6.1. Alcance

Este proyecto tendrá como alcance el diseño, implementación y evaluación de un modelo metodológico optimizado de CI/CD para una aplicación ERP, con el fin de reducir los tiempos de compilación y despliegue, así como el consumo de recursos. El modelo metodológico se basa en la modularización del sistema, compilación selectiva y el despliegue desacoplado, los cuales se materializan mediante la configuración de pipelines CI/CD que permiten que cada subaplicación del ERP sea construida y desplegada de manera autónoma, mejorando la rapidez, eficiencia y control en la entrega de actualizaciones al cliente.

Más allá del caso práctico específico, este trabajo busca establecer un método replicable que pueda ser aplicado en otros sistemas empresariales con características similares. En este sentido, el aporte no se limita únicamente a la mejora puntual de un ERP particular, sino a la formulación de un modelo metodológico que describe cómo diagnosticar los cuellos de botella de un pipeline existente, definir estrategias de modularización y aplicar mecanismos de compilación selectiva y despliegue desacoplado.

De esta manera, el proyecto no solo contribuye a la optimización del flujo de trabajo en un contexto concreto, sino que también propone un marco metodológico escalable y adaptable que puede ser utilizado como referencia en la modernización de pipelines CI/CD para sistemas ERP y otras aplicaciones empresariales de gran tamaño, haciendo posible una mayor flexibilidad, facilidad de mantenimiento y adaptabilidad a los cambios en el entorno tecnológico y de negocio.

6.2. Diseño

La presente investigación se clasifica como un estudio descriptivo, según Sampieri et al. (2014). Este tipo de estudio es adecuado, ya que busca describir, analizar y aplicar las estrategias que optimizan la compilación y despliegue de aplicaciones modulares con el objetivo de identificar el impacto de la modularización y optimizar el proceso de CI/CD en aplicaciones ERP. La ejecución se basa en la configuración de un entorno de pruebas en el que el sistema ERP monolítico de referencia será descompuesto en módulos independientes. Para ello, se utilizarán herramientas como GitLab CI/CD para la implementación de pipelines automatizados, Docker para la contenedorización de cada módulo y Helm para el despliegue en un clúster de Kubernetes. Los datos obtenidos del tiempo de compilación, uso de recursos y estabilidad del sistema en cada etapa serán recolectados y analizados mediante métricas cuantitativas. Estos resultados permitirán comparar la eficiencia y agilidad en el despliegue antes y después de la modularización, permitiendo extraer conclusiones sobre la efectividad del método propuesto.

6.3. Para el desarrollo de la parte informática

Para alcanzar los objetivos planteados, se utilizará la metodología ágil SCRUM, que ayuda a trabajar progresivamente los requisitos necesarios para la implementación. Esta metodología permitirá implementar y optimizar de manera continua los mecanismos utilizados, avanzando gradualmente hasta alcanzar el objetivo final.

Este trabajo de investigación abordará la optimizacion de los procesos de CI/CD para una aplicación ERP monolítica desarrollada y utilizada en la empresa "Smart Cities Perú" tomando los siguientes aspectos:

- Comprensión del ámbito del estudio y definición de objetivos: Se identificarán las fuentes de información más relevantes y se establecerán los límites y objetivos de lo que se desea lograr.
- Se descompondrán los módulos del ERP, el cual permitirá que cada una de las subaplicaciones pueda ser compilado y desplegado de manera autónoma.
- Se configurará un pipeline de CI/CD que detecte cambios a nivel de módulos trabajados para que solo estos sean compilados y luego se realice la construcción con Docker y el despliegue usando Kubernetes, con Helm para la gestión de versiones y despliegues.
- Se evaluará el impacto de la modularización y optimización del pipeline basándonos en tiempos de compilación, despliegue y el consumo de recursos en comparación con la arquitectura monolítica usada.
- Se considerará como la modularización facilita el manteminiento y la escalabilidad del ERP para las actualizaciones y el desarrollo de nuevos módulos, permitiendo establecer recomendaciones para la adopción de esta metodología en otros sistemas similares.

Capítulo 2

Marco teórico

1. Antecedentes

1.1. Antecedentes internacionales

Ferrara, D., Rodríguez, E., Monfroglio, R., Robles, M., Díaz Lapérgola, M. A., Nahuel, L. (2023), *Tecnología colaborativa para investigadores: implementación y optimización de un repositorio GitLab para gestión de la configuración y versionado.* XXIX Congreso Argentino de Ciencias de la Computación, CACIC 2023.

Ferrara et al. (2023) describen la implementación de un servidor Gitlab en infraestructura local como solución a problemas de descentralización, pérdida de información y deficiencias en el control de versiones en proyectos de investigación. La propuesta permite centralizar el versionado de código, la gestión de incidencias y el uso de pipelines de integración continua para automatizar procesos.

La metodología aplicada incluye la identificación del problema, la selección e instalación de GitLab en un servidor propio y su evaluación operativa. Los resultados muestran mejoras en el control de versiones, la trazabilidad de cambios y la coordinación entre usuarios, además de facilitar la automatización de tareas mediante GitLab CI/CD.

Conclusiones:

- La centralización del código y la gestión de proyectos mediante GitLab mejora la trazabilidad, el control de versiones y la colaboración entre equipos.
- La incorporación de pipelines CI/CD sienta las bases para automatizar procesos de compilación y despliegue, incrementando la eficiencia operativa del desarrollo de software.

Comentario: Este antecedente valida la importancia de GitLab y CI/CD como base para la automatización del ciclo de vida del software, sobre la cual la presente investigación incorpora técnicas de modularización y optimización para reducir los tiempos de compilación y despliegue.

Pérez J, (2023), *Integración Continua y Despliegue Continuo de una aplicación*, Universidad de Alicante. Departamento de Lenguajes y Sistemas Informáticos, España.

Conclusiones:

- Pérez López (2023) demostró que la automatización completa del pipeline CI/CD mediante Jenkins, Docker y Kubernetes permite reducir los tiempos de despliegue y aumentar la confiabilidad del proceso, al eliminar intervenciones manuales y ejecutar pruebas y despliegues de forma continua. El estudio evidencia que la integración de estas herramientas mejora la velocidad de entrega y la estabilidad del software, lo que respalda el uso de arquitecturas contenerizadas y pipelines automatizados como base para optimizar sistemas empresariales.

Comentario: El trabajo presentado por Pérez López (2023) será de gran utilidad para la implementación de la infraestructuras de CI/CD en el proyectos, dado que proporciona un marco práctico y probado con tecnologías actuales como Jenkins, Docker y Kubernetes. La experiencia descrita puede servir como base teórica para la creación de entornos de automatización, contribuyendo a la investigación en áreas como DevOps y el uso de contenedores.

Velepucha, V. y Flores, P. (2020). "Monoliths to Microservices - Migration Problems and Challenges: A SMS". En Proceedings of the Escuela Politécnica Nacional, Facultad de Ingeniería de Sistemas. Quito, Ecuador.

En el estudio presentado por Velepucha and Flores (2021) analizan la migración de arquitecturas monolíticas hacia microservicios, indicando que, si bien el modelo monolítico facilita el desarrollo inicial, se convierte en una limitación conforme el sistema crece, afectando la escalabilidad, la velocidad de entrega y la adopción de nuevas tecnologías.

Por otro lado, los microservicios proporcionan mayor flexibilidad, escalabilidad y capacidad de actualización tecnológica; sin embargo, introducen una mayor complejidad en la gestión de pruebas, la orquestación y la coordinación de múltiples subaplicaciones, por lo que su adopción requiere un análisis previo de los objetivos y capacidades de la organización.

Conclusiones:

- La arquitectura monolítica restringe la escalabilidad y la rapidez de entrega a medida que el sistema evoluciona, mientras que los microservicios incrementan la agilidad y la capacidad de actualización, a costa de una mayor complejidad operativa.
- La gestión de múltiples subaplicaciones en arquitecturas de microservicios impacta significativamente en los tiempos de despliegue, haciendo indispensable el uso de mecanismos de orquestación y automatización.

Comentario: Este antecedente sustenta la necesidad de optimizar los procesos de compilación y despliegue en arquitecturas modulares, ya que la complejidad propia de los microservicios puede generar cuellos de botella si no se implementan estrategias adecuadas de automatización y despliegue desacoplado, como las planteadas en la presente investigación.

Vayghan, L. A., Saied, M. A., Toeroe, M., Khendek, F. (2018). Deploying microservice based applications with Kubernetes: Experiments and lessons learned. IEEE, Estados Unidos.

En el trabajo de Abdollahi Vayghan et al. (2018) evalúan experimentalmente la disponibilidad de aplicaciones basadas en microservicios desplegadas con Kubernetes en una nube privada. El análisis se realiza midiendo métricas de reacción, reparación, recuperación y tiempo de indisponibilidad ante fallos de pods y nodos, utilizando Kubernetes sobre tres máquinas virtuales y un servicio de streaming como caso de estudio.

Los autores simulan dos tipos de fallos: fallos administrativos internos de Kubernetes y fallos externos reales, repitiendo cada escenario múltiples veces. Los resultados evidencian que Kubernetes responde rápidamente ante fallos internos, mientras que los fallos externos, especialmente la caída de nodos, generan tiempos de indisponibilidad significativamente mayores.

En fallos de pod, el tiempo de indisponibilidad fue cercano a 1.5 segundos para fallos administrativos y superior a 30 segundos para fallos externos. En fallos de nodo, el tiempo de indisponibilidad fue de aproximadamente 1.5 segundos para fallos administrativos y de hasta 5 minutos para fallos externos, debido a los tiempos de detección y reprogramación por defecto de Kubernetes.

Conclusiones:

- Kubernetes, en su configuración por defecto, no garantiza alta disponibilidad frente a fallos externos, ya que la caída de un nodo puede generar tiempos de indisponibilidad prolongados.
- Los mecanismos de autocuración son eficaces ante fallos internos, pero requieren configuración adicional para reducir los tiempos de recuperación en escenarios reales.

Comentario: Este antecedente evidencia que, si bien Kubernetes facilita el despliegue de microservicios, su configuración por defecto introduce latencias críticas en la recuperación ante fallos, lo que justifica la necesidad de optimizar arquitecturas CI/CD y estrategias de despliegue desacopladas para mejorar la disponibilidad y eficiencia operativa del sistema.

Stricker, R., Müssig, D. y Lässig, J. (2018). "Microservices for Redevelopment of Enterprise Information Systems and Business Processes Optimization". En Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018).

El estudio presentado por Stricker et al. (2018) analizan la transición de sistemas empresariales monolíticos hacia arquitecturas de microservicios en la nube, considerando tanto los beneficios tecnológicos como su efecto en la gestión de servicios bajo el marco ITIL. El estudio muestra que la alta modularidad de los microservicios y su despliegue mediante contenedores incrementan la disponibilidad, la escalabilidad y la rapidez de implementación, facilitando una operación más flexible y eficiente de los sistemas empresariales.

El estudio evidencia que la adopción de microservicios permite optimizar métricas ITIL como MTRS (Mean Time to Restore Service), MRTT (Mean Request for Change Turnaround Time) y TRD (Total Release Downtime), debido a que los servicios son pequeños, se reinician rápidamente y pueden desplegarse de forma independiente, reduciendo el impacto y duración de las interrupciones.

Conclusiones:

- La arquitectura de microservicios reduce significativamente el tiempo de restauración del servicio (MTRS) y el tiempo total de indisponibilidad (TRD), al permitir reinicios y despliegues rápidos de componentes aislados.
- La independencia de los servicios y la automatización de despliegues permiten disminuir el tiempo de implementación de cambios (MRTT) y habilitar ciclos de liberación frecuentes, mejorando la agilidad operativa.

Comentario: Este antecedente respalda que la modularización y los despliegues independientes permiten reducir tiempos de indisponibilidad y acelerar la entrega de cambios, principios que la presente investigación aplica y cuantifica mediante un pipeline CI/CD optimizado para aplicaciones ERP modulares.

2. Bases teóricas

2.1. Metodología SCRUM

Scrum es una metodología ágil que se utiliza para gestionar y desarrollar proyectos, especialmente en entornos de incertidumbre o cambio constante. Se basa en el trabajo en equipo, la colaboración y la entrega incremental de valor a través de ciclos cortos llamados sprints. Su objetivo es adaptarse rápidamente a los cambios y mejorar continuamente el producto y el proceso. Proporciona roles, eventos y artefactos que organizan el trabajo de manera flexible y eficiente.

"Scrum, metodología ágil que ha venido tomando cada vez más fuerza convirtiéndose en una de las favoritas al ser elegida marco de trabajo en proyectos de ingeniería de software." –Hernández Salazar and Beltrán (2022)

2.2. ERP

Es un software de gestión empresarial integrado diseñado para centralizar y automatizar los procesos y flujos de información de las principales áreas de una organización, tales como finanzas, recursos humanos, ventas, etc, utilizando una base de datos unificada que permite la coherencia y disponibilidad de los datos en tiempo real.

En el contexto de esta investigación, se considera que un ERP alcanza un "tamaño considerable" cuando su arquitectura modular y los procesos de integración asociados generan una complejidad técnica significativa (por ejemplo, múltiples módulos interdependientes, altos volúmenes de datos y procesos de compilación extensos), lo cual impacta directamente en métricas como el tiempo de compilación e implementación en los pipelines de integración continua.

A diferencia del software pequeño o de tamaño mediano, el software grande o a medida implica una inversión alta, mayor riesgo y requiere una planificación más detallada de su infraestructura y mantenimiento

"Los sistemas ERP se encuentran entre los tipos de sistemas de información más complejos dentro del ámbito de TI." – Schlichter et al. (2021).

2.3. Arquitectura monolítica

Una arquitectura monolítica es un modelo de desarrollo de software tradicional que utiliza un código base para realizar varias funciones empresariales. Todos los componentes de software de un sistema monolítico son interdependientes debido a los mecanismos de intercambio de datos dentro del sistema. Modificar la arquitectura monolítica es restrictivo y lleva mucho tiempo, ya que los pequeños cambios afectan a grandes áreas del código base AWS (2024).

Este enfoque resulta adecuado en etapas iniciales de proyectos, ya que facilita la implementación y el despliegue, al concentrar todo en una sola unidad. Entre sus principales características se encuentran:

- Fácil de desarrollar y desplegar en las primeras etapas de desarrollo.
- Los cambios en una parte de la aplicación requieren compilar y desplegar toda la aplicación.
- Escalabilidad de tipo vertical, es decir, se limita a aumentar recursos de un único servidor.

Si bien esta arquitectura sigue siendo útil para aplicaciones pequeñas o con requerimientos poco cambiantes, su rigidez genera problemas en entornos empresariales de gran escala, ya que el proceso de despliegue continuo se vuelve lento e ineficiente. Esto es especialmente relevante en el caso de sistemas ERP, donde múltiples áreas de negocio requieren actualizaciones frecuentes y específicas.

2.4. Arquitectura modular

La arquitectura modular surge como respuesta a las limitaciones del modelo monolítico. En este enfoque, la aplicación se divide en módulos independientes que, aunque integrados en un mismo sistema, pueden desarrollarse, compilarse y desplegarse de forma relativamente autónoma. Este diseño promueve la separación de responsabilidades y la reutilización de código. Entre sus características se incluyen:

- Favorece la separación de responsabilidades
- Requiere una configuración adicional en la manera de desarrollo y despliegue de la aplicación
- Permite la reutilización de código entre distintas partes del código de manera óptima.

Aunque implica un esfuerzo adicional en la gestión de dependencias y en la configuración de pipelines de integración y despliegue, este modelo ofrece ventajas significativas en cuanto a escalabilidad horizontal, mantenibilidad y velocidad de entrega de nuevas funcionalidades. Para un ERP empresarial, la modularización permite que áreas críticas (como farmacia, seguros o recursos humanos) puedan evolucionar sin necesidad de impactar a todo el sistema.

Al diseñar aplicaciones con arquitectura modular y aplicar los principios de los microservicios, los desarrolladores pueden crear sistemas más resilientes a los cambios y más fáciles de gestionar – Medium.com (2024).

2.5. Comparación entre arquitecturas

La Tabla 2.1 resume las principales diferencias entre la arquitectura monolítica y la modular, resaltando sus implicancias en el desarrollo y en los procesos de CI/CD:

Tabla 2.1: Tabla de diferencias entre la arquitectura monolítica y la modular

Característica	**Monolítica**	**Modular/Microservicios**
Escalabilidad	Vertical	Horizontal
Tiempo de despliegue	Lento	Rápido
Mantenimiento	Complejo	Más flexible
Complejidad inicial	Baja, fácil de implementar	Alta, requiere configuración adicional
Contexto de uso	Aplicaciones pequeñas o estables	Sistemas grandes y dinámicos

Este contraste resulta clave para el presente estudio, dado que la optimización de tiempos de compilación y despliegue depende en gran medida del tipo de arquitectura. Mientras que el enfoque monolítico restringe la agilidad en los procesos de CI/CD, la modularización abre la posibilidad de aplicar técnicas de compilación selectiva y despliegue independiente, que constituyen la base de la propuesta metodológica de esta tesis.

3. CI/CD

Integración Continua y Despliegue Continuo es una practica en el proceso de desarrollo que automatiza el proceso de integración, pruebas y despliegues en un entorno de desarrollo o producción.

La Integración Continua (CI) y el Despliegue Continuo (CD) constituyen prácticas fundamentales dentro de DevOps, ya que permiten automatizar la integración, pruebas y despliegue de software, reduciendo riesgos de errores manuales y aumentando la frecuencia de entrega de nuevas funcionalidades. En el caso de aplicaciones empresariales modulares, como los ERP, su adopción se justifica por la necesidad de agilidad y confiabilidad en la entrega de cambios.

Entre sus principales ventajas destacan la reducción del tiempo de integración, la detección temprana de errores y la estandarización de procesos. Sin embargo, su implementación implica desafíos, como la necesidad de infraestructura adecuada, la curva de aprendizaje de los equipos y un esfuerzo inicial considerable en sistemas a gran escala.

En esta investigación, CI/CD se plantea como la estrategia base para optimizar los pipelines de compilación y despliegue, utilizando GitLab CI/CD como herramienta principal, integrando el empaquetado con Docker y automatizando despliegues. De esta forma, se busca comparar los tiempos de despliegue antes y después de la optimización propuesta, evidenciando reducciones significativas frente a los procesos manuales.

"During automated deployments, deployment times were significantly reduced compared to manual deployments, especially for iterative deployments. Manual deployments were time-consuming because they required human intervention to perform each step." Hyun et al. (2024).

3.1. Gitlab

"GitLab es un gestor de repositorios Git que permite que los equipos colaboren en códigos informáticos. Está escrito en Ruby y Go, y fue creado en 2011 por Dmitriy Zaporozhets y Valery Sizov". – Datascientest.com (2022).

GitLab no solo funciona como gestor de repositorios Git, sino que además integra herramientas de automatización que lo convierten en una plataforma completa para implementar CI/CD. En el contexto de este trabajo, su elección se justifica porque centraliza el control del código y facilita la integración de pipelines de compilación y despliegue. Sin embargo, su configuración inicial y la administración de runners pueden requerir conocimientos adicionales de infraestructura.

En la presente investigación se utilizará GitLab como herramienta base para gestionar el repositorio de código y los pipelines de CI/CD, permitiendo medir los tiempos de compilación y despliegue antes y después de aplicar la optimización propuesta.

3.2. Gitlab Runner

GitLab Runner es una aplicación que funciona con GitLab CI/CD para ejecutar trabajos en una canalización – GitLab Inc. (2024).

GitLab Runner es una herramienta que actúa como ejecutor de trabajos definidos en los pipelines de GitLab CI/CD. Es responsable de recibir y ejecutar las instrucciones contenidas en los archivos de configuración `.gitlab-ci.yml`.

El uso de este componente permite aislar y automatizar procesos como compilación, pruebas y despliegues, asegurando consistencia en diferentes entornos.

Su principal ventaja es la flexibilidad para ejecutarse en distintos entornos, lo que facilita la escalabilidad del pipeline. Como limitación, requiere una adecuada gestión de recursos, ya que múltiples trabajos en paralelo pueden afectar el rendimiento si la infraestructura es limitada.

Para este proyecto de tesis los Runner serán clave para medir la ejecución modularizada de los pipelines y evaluar la reducción en tiempos de compilación y despliegue, comparándolos con la ejecución monolítica tradicional.

4. Docker

Docker es una plataforma de código abierto que permite a los desarrolladores crear, implementar, ejecutar, actualizar y gestionar aplicaciones en contenedores. – ibm.com (2024)

Su relevancia en el contexto de este trabajo radica en que facilita la estandarización de entornos para los módulos del ERP, evitando inconsistencias entre desarrollo, pruebas y producción. Además, al ser más ligero que una máquina virtual, optimiza el consumo de recursos, lo cual es un factor clave en la reducción de los tiempos de compilación y despliegue que busca esta investigación. Sin embargo, también introduce retos relacionados con la seguridad y la correcta gestión de imágenes, lo cual requiere el uso de registros privados y prácticas de versionado controladas.

- Portabilidad: se garantiza que las aplicaciones funcionen de forma idéntica en cualquier infraestructura.
- Aislamiento: cada contenedor opera de manera independiente, evitando conflictos entre dependencias.

4.1. Contenedores

Según el sitio oficial docker.com (2025) "Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de forma rápida y fiable entre entornos informáticos"

Los contenedores permiten empaquetar aplicaciones junto con sus dependencias, garantizando que se ejecuten de forma uniforme en cualquier entorno. En este proyecto resultan esenciales porque aseguran que los módulos del ERP puedan desplegarse de manera independiente y reproducible dentro del pipeline CI/CD.

Entre sus ventajas destacan la portabilidad, el aislamiento y la eficiencia en el uso de recursos frente a las máquinas virtuales. Como limitación, requieren una gestión cuidadosa en aspectos de seguridad y almacenamiento, especialmente cuando se manejan múltiples contenedores en producción.

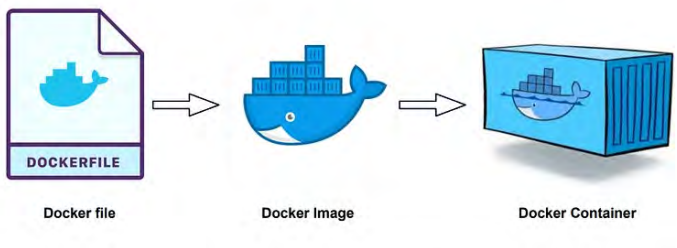
En este trabajo de tesis se utilizará contenedores para encapsular cada módulo del ERP, facilitando la compilación y despliegue independiente dentro del pipeline optimizado. Esto permitirá evaluar mejoras en rapidez y consumo de recursos respecto al enfoque monolítico.

4.2. Imagen Docker

Una imagen docker es una plantilla de solo lectura en el que se define todo lo que necesita un contenedor docker para ejecutarse tales como el sistema operativo, dependencias, archivos y configuración de la aplicación. Una imagen se crea a partir de un archivo Dockerfile, que contiene las instrucciones para construirla, y una vez creada, puede ser ejecutada como múltiples contenedores, esta representación se puede ver en la Figura 2.1.

"Una imagen de Docker es una instantánea o un esquema de las bibliotecas y dependencias necesarias dentro de un contenedor para que se ejecute una aplicación". – amazon.com (2024)

Figura 2.1: Esquema de construcción de un contenedor Docker



Fuente: medium.com (2024)

Entre sus ventajas destacan la reutilización y versionamiento, lo que facilita mantener un historial de cambios en los entornos de ejecución. Como limitación, el tamaño de las imágenes puede crecer si no se optimizan, lo que impacta en los tiempos de construcción y despliegue.

En el método propuesto, los módulos del ERP que se convertirán en subaplicaciones serán construidos a partir de imágenes Docker personalizadas, lo que permitirá comparar la eficiencia en tiempos de construcción y despliegue frente al enfoque monolítico tradicional.

4.3. Registro de contenedores

Un registro de contenedores es un repositorio de datos (o varios repositorios) que almacena imágenes de contenedores para su almacenamiento y acceso. – ibm.com (2025)

Su principal ventaja es que permite un acceso eficiente y controlado a las imágenes versionadas en entornos de desarrollo, pruebas y producción, asegurando y optimizando los flujos de trabajo CI/CD. Una representación de las imágenes y el registro mencionado se evidencia en la Figura 2.2 mostrada a continuación.

Figura 2.2: Representación de un registro de imágenes

5. Kubernetes

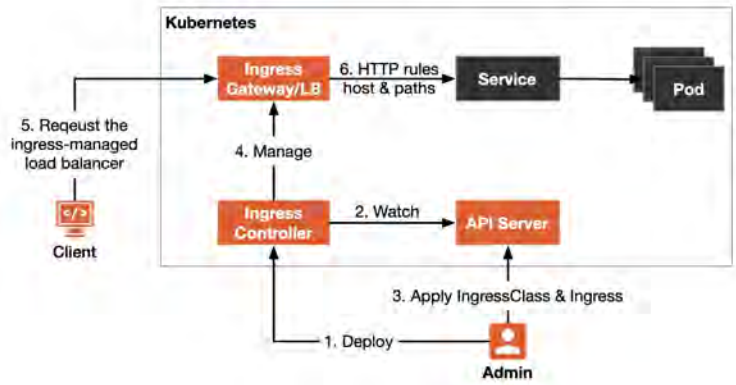
Según el sitio oficial kubernetes.io (2024), “Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios”, Kubernetes permite la automatización del despliegue, escalado y gestión de aplicaciones en contenedores.

Kubernetes se ha consolidado como la herramienta estándar para la orquestación de contenedores en entornos modernos. Su uso es clave cuando se gestionan aplicaciones distribuidas y de gran escala, ya que automatiza tareas críticas como la asignación de recursos, la recuperación ante fallos y la escalabilidad horizontal. En este proyecto, se justifica su inclusión como soporte para la gestión de los módulos del ERP en un entorno productivo más robusto.

Como limitación, Kubernetes introduce una curva de aprendizaje considerable y puede implicar un consumo adicional de recursos, lo cual representa un reto en entornos de menor tamaño o con recursos limitados.

El método propuesto utiliza Kubernetes para evaluar la eficiencia del despliegue de módulos ERP en un clúster, analizando su escalabilidad, uso de recursos y capacidad de recuperación frente a fallos, en comparación con enfoques tradicionales. La Figura 2.3 muestra el flujo de trabajo de kubernetes.

Figura 2.3: Flujo de trabajo de Kubernetes



Fuente: tetrate.io (2024)

5.1. Clúster

Un clúster es un conjunto de recursos de hardware y software el que se despliegan, gestionan y coordinan aplicaciones en contenedores coordinando múltiples nodos y automatizando tareas como la distribución de cargas, la recuperación de fallos y la escalabilidad de las aplicaciones. Es la estructura fundamental que permite a Kubernetes orquestar y escalar aplicaciones de manera automática.

"Kubernetes has a lot of tools and extensions that make it more useful, in addition to being scalable and available. The Kubernetes API lets developers add their own custom resources and operators to the platform" Oduri (2024).

6. Helm

Helm surge como complemento natural de Kubernetes, funcionando como un gestor de paquetes que simplifica la instalación y actualización de aplicaciones complejas en entornos de contenedores. Su uso se justifica en este proyecto porque permite estandarizar despliegues y reducir la carga operativa, lo cual resulta esencial cuando se busca optimizar procesos CI/CD en arquitecturas modulares.

Entre sus ventajas se encuentra la posibilidad de reutilizar configuraciones mediante charts, lo que facilita la consistencia en los entornos y minimiza errores manuales.

En el método de investigación, Helm se usará para gestionar el despliegue de módulos del ERP sobre Kubernetes, permitiendo comparar la eficiencia y estandarización del proceso frente a configuraciones manuales, y midiendo cómo su uso contribuye a la reducción de tiempos y errores en los despliegues.

"Helm te ayuda a administrar las aplicaciones de Kubernetes - Los Charts de Helm te ayudan a definir, instalar y actualizar incluso la aplicación de Kubernetes más compleja". – helm.sh (2025)

7. Cloud

El cloud, o la "nube", son un conjunto de servicios y recursos informáticos que se ofrecen de manera remota, permitiendo acceder a ellos desde cualquier lugar a través de Internet. El cloud permite que las aplicaciones y servicios desplegados con Kubernetes se ejecuten en grandes infraestructuras gestionadas por proveedores como Google Cloud, AWS o Azure.

Kubernetes en el cloud facilita el escalado y la administración de aplicaciones sin necesidad de preocuparse por la infraestructura física ya que al ser escalable, y controlada, permite a los equipos enfocarse en el desarrollo sin preocuparse por la infraestructura de fondo.

El cloud computing es la disponibilidad de recursos de computación bajo demanda (como almacenamiento e infraestructura) como servicios a través de Internet. Elimina la necesidad de que las personas y las empresas gestionen sus propios recursos físicos y paguen solo por lo que utilicen. – cloud.google.com (2025)

Figura 2.4: Kubernetes en la nube

Fuente: xataka.com (2019)

8. Angular

Angular es un framework para el desarrollo de aplicaciones web escalables, creado por Google permitiendo construir interfaces interactivas y dinámicas de manera organizada y eficiente. Angular facilita el trabajo con HTML, CSS y JavaScript, ofreciendo herramientas que ayudan a organizar el código en componentes que forman parte de la interfaz, los cuales podrán ser reutilizables.

Este framework se utiliza en el ERP que se tomará como caso práctico. Emplearemos las herramientas que permiten la descomposición de los módulos en bibliotecas y subaplicaciones. Una de las principales ventajas de este framework es su alta robustez y el hecho de que es ampliamente utilizado por importantes empresas a nivel mundial.

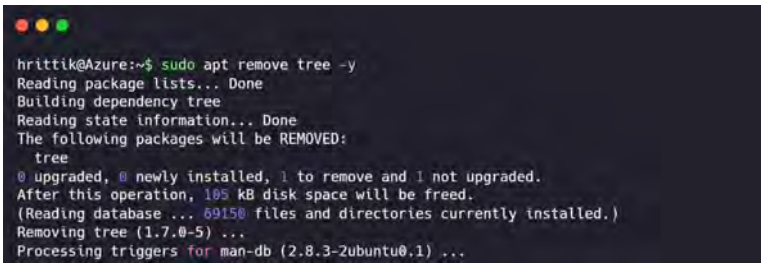
"Con Angular, aprovechas una plataforma escalable desde proyectos de un solo desarrollador hasta aplicaciones empresariales. Lo mejor de todo es que el ecosistema Angular está compuesto por un grupo diverso de más de 1,7 millones de desarrolladores, autores de bibliotecas y creadores de contenido". – angular.io (2025)

9. Bash

"Un shell es una interfaz basada en texto que le permite comunicarse con su computadora". – w3schools.com (2025)

En bash podemos escribir scripts son muy útiles para realizar tareas repetitivas, como compilar código, ejecutar pruebas y desplegar aplicaciones, todo de forma automática. Estos son comúnmente usados en pipelines porque permite escribir comandos de manera sencilla y directa, ideal para tareas que se ejecutan en sistemas operativos basados en Unix, como Linux.

Figura 2.5: Bash en sistemas Unix



Fuente: earthly.dev (2023)

9.1. YQ

Según el sitio oficial, "jq es un procesador JSON de línea de comandos ligero y flexible.--jqlang.org (2024).

En este trabajo de tesis, se utilizará para realizar tareas específicas dentro de determinados pasos (steps) de los pipelines.

Capítulo 3

Desarrollo del tema de tesis

1. Metodología propuesta de optimización en CI/CD para aplicaciones modulares

La metodología propuesta en esta investigación tiene como objetivo optimizar los tiempos de compilación y despliegue de aplicaciones empresariales modulares mediante la aplicación de principios de compilación selectiva, construcción paralela de contenedores y despliegue desacoplado.

Esta metodología evalúa el desempeño de un mismo sistema ERP antes y después de aplicar un conjunto de optimizaciones sobre su pipeline de CI/CD, utilizando métricas cuantitativas de tiempo de ejecución.

1.1. Modelo metodológico

La metodología se estructura en cuatro fases secuenciales y dependientes:

- Diagnóstico del pipeline actual.
- Diseño de la arquitectura modular.
- Implementación del pipeline optimizado.
- Evaluación comparativa de resultados.

Cada fase produce salidas medibles que permiten cuantificar el impacto de la optimización sobre los tiempos de compilación, construcción de imágenes y despliegue.

1.2. Fase 1 – Diagnóstico del pipeline actual

Objetivo: Establecer la línea base de desempeño del sistema ERP bajo arquitectura monolítica.

Actividades:

- Ejecución del pipeline CI/CD original en GitLab.
- Registro de tiempos de compilación, construcción de imágenes y despliegue.
- Identificación de módulos afectados por cada cambio de código.

Salida: Tiempos de ejecución del pipeline monolítico por escenario de cambio.

1.3. Fase 2 – Diseño de la arquitectura modular

Objetivo: Definir la estructura que permitirá compilar y desplegar partes del sistema de forma independiente.

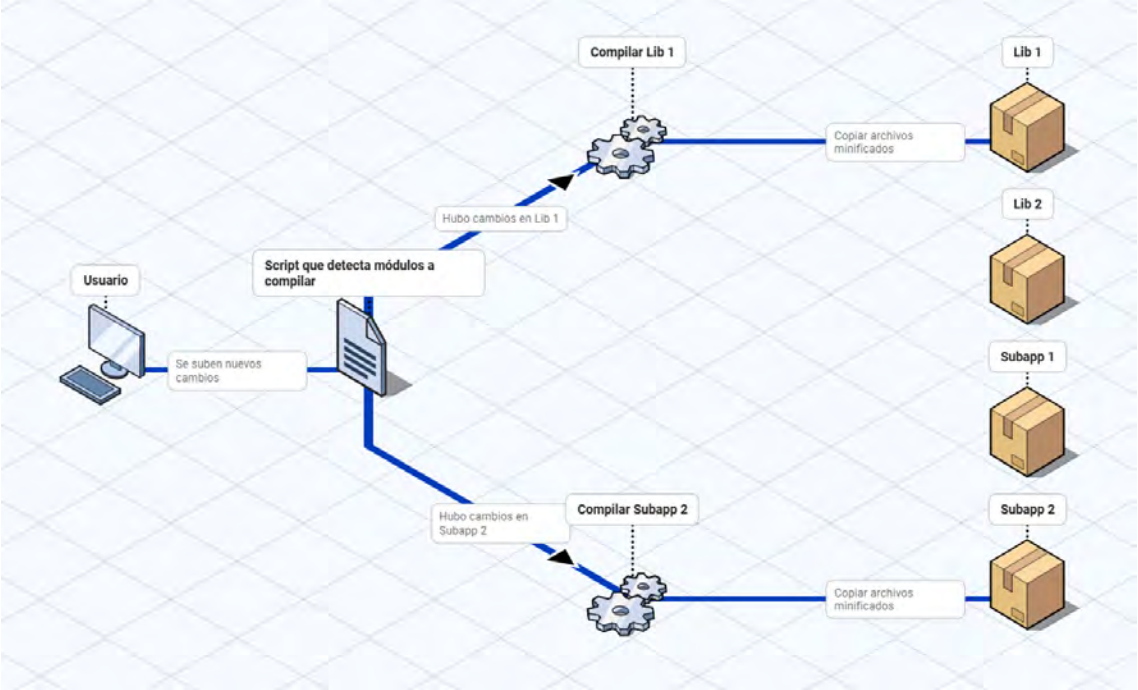
Actividades:

- Descomposición del ERP en librerías Angular reutilizables.
- Definición de subaplicaciones independientes.
- Identificación de dependencias compartidas.
- Asignación de responsabilidades funcionales por módulo.

Salida: Mapa de módulos, subaplicaciones y dependencias que permite identificar qué componentes deben recompilarse ante cada cambio.

Esta fase habilita técnicamente la compilación selectiva y el despliegue desacoplado. Además este enfoque es consistente con los modelos de arquitectura modular descritos por Velepucha and Flores (2021), quienes destacan que la transición hacia sistemas modulares mejora la mantenibilidad y escalabilidad, aunque introduce complejidades adicionales en términos de orquestación y coordinación entre módulos. Este paso fue determinante para aplicar la metodología de optimización propuesta como se muestra en la Figura 3.1

Figura 3.1: Vista gráfica de compilación selectiva



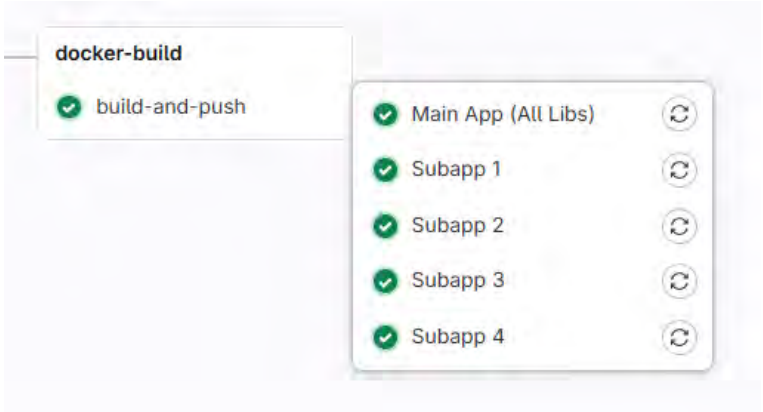
1.4. Fase 3 – Implementación del pipeline optimizado

Objetivo: Aplicar los mecanismos técnicos que permiten reducir los tiempos de ejecución del pipeline.

Actividades:

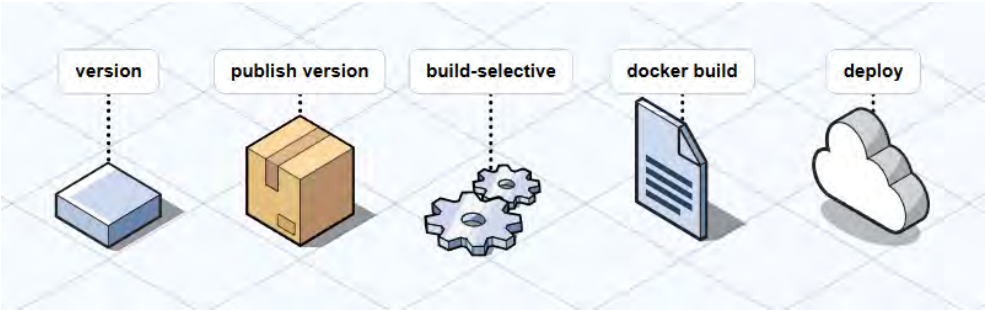
- Compilación selectiva de módulos afectados por un cambio.
- Construcción paralela de imágenes Docker (Ver Figura 3.2).

Figura 3.2: Gráfico de construcción paralela propuesto en Gitlab CICD



Salida: Pipeline CI/CD modular funcional y versiones independientes por subaplicación, ver la Figura 3.3.

Figura 3.3: Vista gráfica del pipeline propuesto



1.5. Fase 4 – Evaluación comparativa

Objetivo: Cuantificar el impacto de la optimización sobre los tiempos de ejecución.

Actividades:

- Ejecución de los mismos escenarios de cambio utilizados en la Fase 1.
- Registro de tiempos del pipeline optimizado.
- Comparación directa con los tiempos del pipeline monolítico.

Salida: Tiempos de compilación, construcción y despliegue del pipeline optimizado y el porcenntaje de reducción de tiempos.

1.6. Instrumentos de medición

Las métricas fueron obtenidas mediante:

- Logs de ejecución de GitLab CI/CD.
- Registros de Docker build.
- Despliegues en Kubernetes vía Helm.
- Historial de pipelines en GitLab.

2. Investigación del ámbito de trabajo

2.1. Análisis del estado actual del ERP

Para iniciar con la implementación, es necesario analizar y estudiar el estado actual del ERP monolítico, las tecnológicas y componentes usados, se identifico el uso de Angular (v15) para el frontend, Go (v20) para el backend, un Runner local para Gitlab para la compilación de la aplicación, Gitlab local como plataforma de manejo de código y Docker con Kubernetes para la contenedorización y orquestación. Debemos revisar como el pipeline integrado se comporta actualmente en el proceso de compilación y despliegue. Ver Figuras 3.4 y 3.5

Figura 3.4: Pipeline del frontend

Figura 3.5: Pipeline del backend

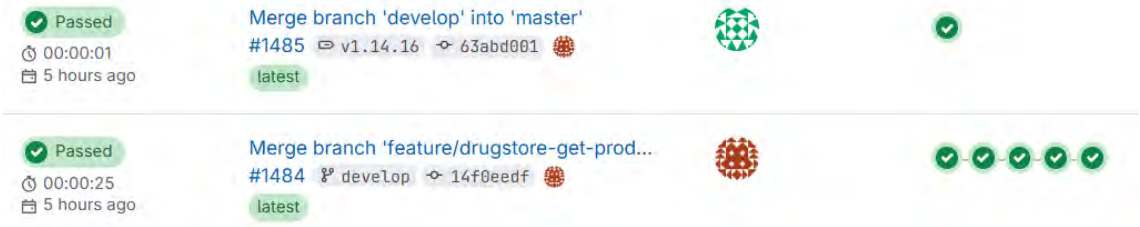


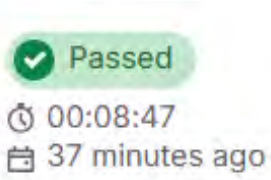
Tabla 3.1: Tabla de tiempos de ejecución de los proyectos del ERP

Tag	App ERP	Minutos
v1.12.105	Frontend	08:47
v1.14.16	Backend	00:25

2.2. Identificación de problemas y oportunidades

En el análisis del proceso de despliegue actual, se observa en la Tabla 3.1 que el tiempo de ejecución del pipeline del Frontend consume considerablemente más tiempo en comparación con el pipeline del Backend. Esta situación impacta negativamente en la eficiencia general, la resolución de errores y la entrega oportuna de nuevas funcionalidades a los clientes.

Figura 3.6: Tiempo de ejecución del pipeline del frontend



Analizando el proceso de despliegue actual en los pipelines existentes, se identificó que la aplicación del Frontend cuenta con una gran cantidad de módulos integrados dentro del ERP en una sola carpeta de trabajo principal como se ve en la figura 3.7, lo que sugiere una oportunidad para descomponer la aplicación y optimizar el manejo de estos módulos. Además, el proceso actual requiere la reconstrucción completa de la aplicación para realizar pruebas en el entorno de desarrollo antes enviar a producción, lo cual añade una carga innecesaria al pipeline, especialmente si no se trabajaron otros módulos.

Figura 3.7: Módulos que compone el ERP actual

En la siguiente sección se presentarán maneras de optimizar la estructura actual del Frontend enfocándonos en mejorar la eficiencia del proceso de compilación y despliegue.

2.3. Aplicación de la recopilación de fuentes de información

Para el desarrollo de esta investigación se consultó documentación oficial de Angular, Docker, Kubernetes y Helm, así como lineamientos y buenas prácticas de CI/CD. Además, se revisaron artículos académicos, guías técnicas y estudios previos sobre la modularización de aplicaciones monolíticas en entornos empresariales, con especial énfasis en implementaciones basadas en Angular. Esta recopilación permitió fundamentar el marco teórico y orientar las decisiones metodológicas en cuanto a modularización, contenedorización y optimización de pipelines.

2.4. Objetivos de optimización

Para mejorar la eficiencia de compilación, se decidió modularizar la aplicación en componentes independientes que se compilen por separado. En Angular, esto puede lograrse mediante el uso de un término denominado librerías, que permiten dividir la aplicación en módulos autónomos. De este modo, cada librería la podemos compilar de forma independiente, a su vez existe el término de sub-aplicaciones el cual funciona de manera similar a las librerías, pero este esta aún más aislada, estaremos procesando y compilando solo aquellas que hayan tenido cambios recientes, usaremos ambos enfoques el cual optimiza significativamente el tiempo de compilación.

3. Descomposición de módulos del ERP

Para optimizar la aplicación ERP, primero se descompondrá en librerías, y luego se estructura en subaplicaciones. Este enfoque, en ambas fases, se permitirá que solo se compilen los módulos que han sido modificados.

3.1. Descomposición inicial en librerías Angular

Se descompondrá la aplicación en librerías independientes, reduciendo significativamente el tiempo de compilación al compilar solo las librerías modificadas. Esta descomposición modular facilita un proceso de construcción más rápido y una mejor organización del código. Para lo cual realizamos los siguientes pasos:

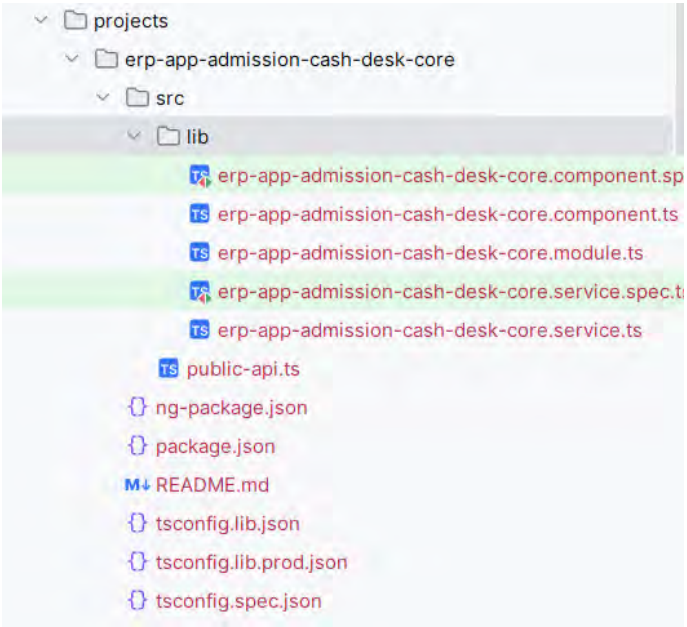
1. **Generación de librería:** Con el comando `ng generate library`, se creó una librería para cada módulo identificado. Cada librería encapsula su lógica de negocio y sus componentes, lo que permite que estos se mantengan independientes de otros módulos.

Figura 3.8: Generación de una librería en Angular

```
usuario@DESKTOP-2CC09U5 MINGW64 /d/dev/macsalud-app-2.2 (develop)
$ ng g library erp-app-admission-cash-desk-core
CREATE projects/erp-app-admission-cash-desk-core/ng-package.json (181 bytes)
CREATE projects/erp-app-admission-cash-desk-core/package.json (236 bytes)
CREATE projects/erp-app-admission-cash-desk-core/README.md (1182 bytes)
CREATE projects/erp-app-admission-cash-desk-core/tsconfig.lib.json (314 bytes)
CREATE projects/erp-app-admission-cash-desk-core/tsconfig.lib.prod.json (240 bytes)
CREATE projects/erp-app-admission-cash-desk-core/tsconfig.spec.json (273 bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/public-api.ts (259 bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/lib/erp-app-admission-cash-desk
bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/lib/erp-app-admission-cash-desk
ts (751 bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/lib/erp-app-admission-cash-desk
65 bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/lib/erp-app-admission-cash-desk
 (467 bytes)
CREATE projects/erp-app-admission-cash-desk-core/src/lib/erp-app-admission-cash-desk
 bytes)
UPDATE angular.json (80517 bytes)
UPDATE tsconfig.json (4544 bytes)
\ Installing packages (npm)...
```

2. **Mover los componentes:** Se movió los componentes pertenecientes al módulo de Admisión del ERP a la librería, exportando la ruta del archivo `.module.ts` en el archivo `public-api.ts` en el cual se exportan todos los servicios, componentes, etc que de quiera exponer para compartirlos con otras librerías.

Figura 3.9: Carpetas que genera una librería



3.2. ¿Qué pasa con dependencias compartidas entre librerías?

Para estructurar las dependencias de manera eficiente y evitar bucles o ciclos entre librerías, se usará la siguiente organización:

- Una librería "shared", el cual contendrá elementos compartidos utilizados en toda la aplicación. Servirá como el punto común para evitar dependencias redundantes entre módulos.
- Se crearán librerías "core" específicas para ciertos módulos complejos. Por ejemplo usaremos el comando `ng g library admission-cash-desk-core` el cual contendrá elementos o recursos compartidos entre las librerías de admisión y farmacia exportando los elementos necesarios a través del archivo "public-api.ts".
- Se respetará el orden de compilación propuesto en la figura 3.18, para evitar compilar librerías sin antes compilar las dependencias que ésta requiere.

Al importar únicamente lo necesario desde las librerías "core", evitamos dependencias directas entre módulos y, a su vez, ciclos de dependencia.

3.3. Descomposición en subaplicaciones Angular

Los módulos o librerías de la aplicación pueden convertirse en subaplicaciones independientes, lo que permite desarrollarlas, compilarlas y lo mas importante, desplegarlas de forma autónoma.

Con este enfoque, el equipo solo necesita iniciar la subaplicación específica en la que está trabajando, mientras que las demás librerías de la aplicación principal permanecen en modo de escucha para detectar y reflejar cambios. Esto hace que el proceso de desarrollo sea más ágil, ya que no es necesario compilar toda la aplicación en cada cambio.

Esta arquitectura no solo acelera el desarrollo, sino que también mejora la escalabilidad y el mantenimiento de la aplicación a lo largo del tiempo.

Los módulos seleccionados para esta descomposición son:

- Admisión y Cajas
- Farmacia
- Seguros
- Recursos Humanos
- Scrum

Utilizando en comando `ng generate application` de Angular, cada módulo se convierte en una subaplicación, aislada del código base principal, que ejecuta su propio proceso de compilación y despliegue. Aún así las subaplicaciones permanecen dentro del mismo repositorio de trabajo, pero se configuran en base al archivo "angular.json".

Asignaremos puertos específicos en el que se levantarán estas subaplicaciones en tiempo de desarrollo según la Tabla 3.2 que se muestra a continuación:

Tabla 3.2: Puertos expuestos por cada subaplicación en etapas de desarrollo

Subaplicación	**Puerto expuesto**
Admisión y Cajas	4302
Farmacia	4303
Seguros	4304
Recursos Humanos	4305
Scrum	4306

Ahora en la Figura 3.10 se visualiza la configuración del puerto asignado para la subaplicación de Admisión y Cajas así como su ruta base de acceso, esto servirá como referencia para las demás subaplicaciones trabajadas.

Figura 3.10: Estructura que genera la creación de la subaplicación de admisión y cajas

```
"erp-app-admission": {
  "projectType": "application",
  "schematics": {},
  "root": "projects/erp-app-admission",
  "sourceRoot": "projects/erp-app-admission/src",
  "prefix": "app",
  "architect": {
    "build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
        "outputPath": "dist/erp-app-admission",
        "index": "projects/erp-app-admission/src/index.html",
        "main": "projects/erp-app-admission/src/main.ts",
        "polyfills": [
          "zone.js"
        ],
        "baseHref": "/admission/"
      },
    },
    "serve": {
      "builder": "@angular-devkit/build-angular:dev-server",
      "configurations": {
        "production": {
          "browserTarget": "erp-app-admission:build:production"
        },
      },
      "defaultConfiguration": "development",
      "options": {
        "port": 4302,
        "host": "0.0.0.0"
      }
    }
  }
}
```

4. Ajuste en el Pipeline de CI/CD

4.1. Actual proceso de compilación en CI/CD

En la Figura 3.11 vemos la primera parte del pipeline, en este se especifican las variables globales y **Stages** que se ejecutarán en el pipeline.

Figura 3.11: Variables y stages del pipeline actualmente usado

```yaml
variables:
  REPO: gitlab.smartcitiesperu.com
  GROUP: mac-salud
  PROJECT: erp

stages:
  - version
  - version-publish
  - build
  - deploy
```

En el proceso de ejecución del stage denominado version, se genera la siguiente versión del ERP, la cual se almacenará en un archivo de entorno llamado build.env como se muestra en la Figura 3.12. Este archivo permite que la versión generada esté disponible como variable de entorno en los siguientes stages del pipeline, facilitando la gestión de versiones en todo el proceso de integración y despliegue continuo.

Figura 3.12: Stage que construye la nueva versión para la aplicación

```yaml
version:
  stage: version
  image: golang:latest
  only:
    - develop
    - master
  script:
    - touch build.env
    - chmod +x cicd/gen-version.sh
    - VERSION=$(bash cicd/gen-version.sh | tr -d '\n')
    - echo "$CI_COMMIT_BRANCH - $VERSION"
    - |
      if [ "${VERSION:0:1}" != "v" ]; then
        echo "Cannot generate version, result: $VERSION"
        exit 1
      fi
    - echo "VERSION=$VERSION" >> build.env
    - echo "$CI_COMMIT_BRANCH - $VERSION"
  tags:
    - angular
  artifacts:
    paths:
      - build.env
    reports:
      dotenv: build.env
```

Como se observa en la Figura 3.13, este script genera la siguiente versión del ERP en función de la última generada, tomando en cuenta la rama actual en la que se está ejecutando el pipeline. Esto permite simplificar el archivo .yaml del pipeline al reducir la cantidad de scripts necesarios.

Figura 3.13: Script que genera la versión consecutiva de una aplicación

```bash
#!/bin/bash

# Get the current branch name
current_branch=$CI_COMMIT_BRANCH

# Get the latest tag from the repository and clean newline and whitespace characters
latest_tag=$(git describe --tags $(git rev-list --tags --max-count=1) | sed 's/v//g')

# Parse version components from the latest tag
IFS='.' read -r -a version_components <<< "$latest_tag"
major=${version_components[0]}
minor=${version_components[1]}
patch=${version_components[2]}

# Function to increment version number based on the branch
increment_version() {
    local major=$1
    local minor=$2
    local patch=$3
    local branch=$4
    local new_minor

    if [ "$branch" == "develop" ]; then
        next_version="v$major.$minor.$((patch + 1))"
    elif [ "$branch" == "master" ]; then
        next_version="v$major.$((minor + 1)).0"
    else
        # Unknown branch, do not increment version
        next_version="v$major.$minor.$patch"
    fi

    # Print the new version
    echo "$next_version"
}


# If there is no previous tag, set initial version as v1.0.0
if [ -z "$latest_tag" ]; then
    next_version="v1.0.0"
else
    # Calculate the new version according to the specified rules
    next_version=$(increment_version "$major" "$minor" "$patch" "$current_branch")
fi

# Pint the next tag
echo "$next_version"
```

El siguiente stage, llamado version-publish como se observa en la Figura 3.14, publica la versión generada en el paso anterior (Figura 3.12) en el repositorio GitLab mediante la creación y envío de una etiqueta (Tag) con el número de versión correspondiente. Este proceso asegura que cada versión quede registrada en el control de versiones.

Figura 3.14: Stage de publicación de un tag

```
version-tag:
  stage: version-publish
  image: golang:latest
  only:
    - develop
    - master
  script:
    - git remote remove gitlab_origin || true
    - git remote add gitlab_origin
https://oauth2:_7tsqKEkKje2xPZRUqLT@gitlab.smartcitiesperu.com/edwar/erp.git
    - git tag $VERSION
    - git push gitlab_origin $VERSION -o ci.skip
  tags:
    - angular
```

Fuente: Elaboración propia

Ahora en la siguiente figura presentada a continuación (Figura 3.15) se observa el proceso que sigue cada vez que el stage build-dev se ejecuta.

1. Se verifica si ya existe una carpeta llamada "node modules"(donde se almacenan todos los paquetes instalados necesarios para la aplicación) para evitar reinstalar paquetes que ya están presentes
2. Se realiza una compilación completa de la aplicación sin distinción de cambios específicos en los módulos.
3. Se construye la imagen Docker de la aplicación y se sube a un registro privado de imágenes especificando la versión de la imagen contruida.

Figura 3.15: Job de compilación y construcción actual de la aplicación ERP

```
build-dev:
  stage: build
  only:
    - master
    - develop
  dependencies:
    - version
  script:
    - source ~/.bashrc
    - ls -1
    - |
      if [ ! -d "$HOME/cache-erp/node_modules" ]; then
        nvm use 18.10.0
        echo -e "\e[33m Building node_modules: \e[0m"
        npm install --legacy-peer-deps
        cp -R node_modules $HOME/cache-erp/
      else
        echo -e "\e[33m Without node_modules: \e[0m"
      fi
    - ls -1
    - ln -s $HOME/cache-erp/node_modules node_modules
    - sed -ie "s/VERSION_BUILD/$VERSION/g" src/index.html;
    - nvm use 18.10.0
    - pwd
    - time node --max_old_space_size=8192 `which npm` run build
    - ls -1
    - |
      echo -e "\e[33m Building docker and push: \e[0m"
      echo "$PROJECT"
      echo "$VERSION"
    - docker build -t $PROJECT:$VERSION -f cicd/docker/Dockerfile .
    - docker tag $PROJECT:$VERSION registry.scp.smartc.pe:5005/$PROJECT:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT:$VERSION
  tags:
    - angular
```

Fuente: Elaboración propia

4.2. Actual proceso de despliegue en CI/CD

En el último paso del pipeline (deploy), como se muestra en la Figura 3.16 , se automatiza el despliegue de la aplicación en Kubernetes. Usando la versión guardada en build.env, se actualiza el archivo values.yaml del Helm Chart para reflejar la nueva versión de la imagen. Luego, mediante conexión SSH, se ejecuta helm upgrade para aplicar los cambios en el entorno erp-dev, asegurando que la aplicación esté actualizada en el clúster de desarrollo.

Figura 3.16: Job de despliegue actual en el entorno de desarrollo

```
deploy:
  stage: deploy
  only:
    - develop
  dependencies:
    - build-dev
  script:
    - source build.env
    - >
      exe_command="
      yq -i '.appv2.image.tag = strenv(VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
      kubectl --kubeconfig=/home/scpadm/.kube/config-local --namespace=erp-dev delete ingress erp-dev-ingress;
      cd $HOME/macsalud-devops/dev/chart;
      helm upgrade erp-dev ./erp-dev --namespace erp-dev --kubeconfig=/home/scpadm/.kube/config-local;
      "
    - ssh -t scpdm@192.168.71.200 -p 22 "$exe_command"
```

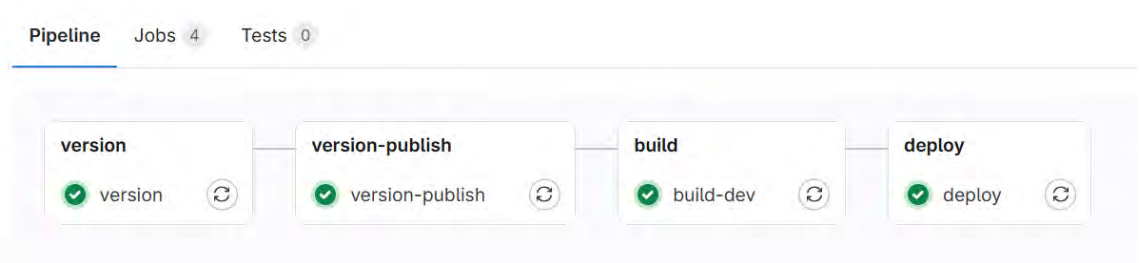
4.3. Pipeline actual

En la Figura 3.17 se muestra la estructura del pipeline configurado inicialmente en GitLab CI/CD antes de aplicar las optimizaciones propuestas. Este pipeline contemplaba cuatro etapas principales:

- **Version:** generación del número de versión de la aplicación.
- **Version-publish:** publicación de la versión en el repositorio correspondiente.
- **Build:** compilación de la aplicación completa, sin distinción entre librerías o subaplicaciones.
- **Deploy:** despliegue de la aplicación resultante en el entorno de pruebas o producción.

Si bien esta configuración permitió establecer un flujo básico de integración y entrega continua, presentaba limitaciones significativas. La principal desventaja radicaba en que el proceso de compilación era secuencial y se realizaba sobre toda la aplicación, lo que generaba largos tiempos de espera (entre 8 y 10 minutos por cada cambio aplicado). Además, no se contemplaba un mecanismo de compilación selectiva ni paralelización de tareas, lo que restringía la eficiencia y escalabilidad del pipeline en un entorno empresarial de gran tamaño.

Figura 3.17: Gráfico del pipeline actual en Gitlab



4.4. Optimización del pipeline para compilación modular

Para optimizar la compilación de las librerías y subaplicaciones del ERP, se realizarán los siguientes pasos:

1. **Detección de cambios:** Tras unir el merge request (solicitud para unir código), el script identifica los archivos que han sufrido cambios usando `GIT DIFF` como se muestra en la figura 3.18.

Figura 3.18: Archivos modificados y módulos del ERP

```
FILES_MODIFIED=$(git diff --name-only $CI_COMMIT_BEFORE_SHA $CI_COMMIT_SHA)

LIBRARIES=(
  "erp-app-core"
  "erp-app-shared"
  "erp-app-rrhh-organizacion-gh-core"
  "erp-app-recursos-humanos-core"
  ...
)

SUBAPPS=(
  "erp-app-admission"
  "erp-app-drugstore"
  "erp-app-insure"
  "erp-app-human-resources"
)
```

2. **Comprobación de módulos minificados en el runner:** Se verifica si los módulos minificados se encuentran en la carpeta `dist`. En caso de que no existan, estos deberán recompilarse nuevamente (ver Figura 3.19).

Figura 3.19: Verificación de la existencia de módulos minificados

```
CHANGED_LIBRARIES=()
CHANGED_SUBAPPS=()

# Comprobar si los módulos están en dist, si no, agregar a changes (LIBRARIES o APP)
for LIB in "${LIBRARIES[@]}"; do
  if [ ! -d "dist/$LIB" ]; then
      CHANGED_LIBRARIES+=("$LIB")
  fi
done

for APP in "${SUBAPPS[@]}"; do
  if [ ! -d "dist/$APP" ]; then
      CHANGED_SUBAPPS+=("$APP")
  fi
done

    done
  fi
done
```

3. **Extracción de cambios en librerias y aplicaciones:** En base al paso anterior, se extraerán las librerías y aplicaciones que necesitan ser recompiladas. En la Figura 3.20 se visualizan los pasos que se realizó para cumplir este paso.

Figura 3.20: Generación de librerías para ser recompiladas

```
CHANGED_LIBRARIES=()
CHANGED_SUBAPPS=()

# Procesar los archivos modificados y agregar los modulos correspondientes
for FILE in $FILES_MODIFIED; do
  if [[ $FILE == projects/* ]]; then
    for SUBAPP in "${SUBAPPS[@]}"; do
      if [[ $FILE == projects/$SUBAPP/* ]]; then
        CHANGED_SUBAPPS+=($SUBAPP)
      fi
    done
    for LIBRARY in "${LIBRARIES[@]}"; do
      if [[ $FILE == projects/$LIBRARY/* ]]; then
        CHANGED_LIBRARIES+=($LIBRARY)
      fi
    done
  fi
done

# Elimina duplicados y ordena
CHANGED_LIBRARIES=($(echo "${CHANGED_LIBRARIES[@]}" | tr ' ' '\n' | sort -u))
CHANGED_SUBAPPS=($(echo "${CHANGED_SUBAPPS[@]}" | tr ' ' '\n' | sort -u))
```

4. **Compilación selectiva:** Únicamente se ejecuta el comando `ng build name_library` (nombre de la librería asignada en el archivo angular.json) o `ng build name_app` (nombre de la aplicación asignada en el archivo angular.json) según corresponda y así compilar la librería o aplicación afectada, generando sus archivos minificados en la carpeta dist (por defecto). Se visualiza este proceso en la Figura 3.21 a continuación.

Figura 3.21: Compilación selectiva de librerías y subaplicaciones

```
# Construye librerias modificadas
for module in  "${LIBRARIES[@]}"; do
    if [[ " ${CHANGED_LIBRARIES[@]} " =~ " ${module} " ]]; then
      echo -e "\e[32mBuilding library: $module\e[0m"
      time ./node_modules/.bin/ng build @scp/$MODULE  # Descomenta para
ejecutar la compilación real
    fi
done


# Construye apps modificadas
if [ ${#CHANGED_SUBAPPS[@]} -gt 0 ]; then
  for SUBAPP in "${CHANGED_SUBAPPS[@]}"; do
    echo -e "\e[33mBuilding app: $SUBAPP\e[0m"
    time ./node_modules/.bin/ng build $SUBAPP
  done
fi
```

5. **Compilación general:** Una vez que los archivos minificados de las librerías específicas están en dist, se realiza una compilación general utilizando `npm run build`. Este proceso es necesario únicamente cuando se han realizado cambios en alguna de las librerías, ya que estas aún forman parte de la aplicación principal. A diferencia de las subaplicaciones, que están desacopladas del núcleo de la aplicación y pueden compilarse de forma independiente. Esto lo podemos observar en la Figura 3.22.

Figura 3.22: Compilación general para unir todas las librerías en una sola aplicación

```
# Solo se ejecuta si no hubo cambios en las subaplicaciones:
if [ ${#CHANGED_LIBRARIES[@]} -gt 0 ]; then
  echo -e "\e[34mBuilding main application: ng build \e[0m"
  time node --max_old_space_size=8192 `which npm` run build
fi
```

Comentario: Gracias a esta estructura, al existir los archivos ya minificados de cada librería, el `ng build` general no recompila las librerías desde cero, sino que toma los archivos ya minificados en dist, lo cual reduce considerablemente el tiempo total de compilación. Esta estrategia modular permite que Angular compile solo las librerías afectadas, acelerando el proceso de integración continua al evitar recompilaciones innecesarias.

Por lo tanto, el stage de **build-dev** quedaría de la siguiente manera (Figura 3.23), el cual ejecuta el script que contiene la lógica de compilación selectiva presentada anteriormente:

Figura 3.23: Nuevo stage de compilación de librerías y subaplicaciones

```
build-dev:
  stage: build
  only:
    - develop
  script:
    - source ~/.bashrc
    - ls -1
    - |
      if [ ! -d "$HOME/cache-erp/node_modules" ]; then
        nvm use 18.10.0
        echo -e "\e[33m Building node_modules: \e[0m"
        npm install --legacy-peer-deps
        cp -R node_modules $HOME/cache-erp/
      else
        echo -e "\e[33m Without node_modules: \e[0m"
      fi
    - ls -1
    - ln -s $HOME/cache-erp/node_modules node_modules
    - ln -s ~/cache-macsalud-v2/dist-stg dist
    - nvm use 18.10.0
    - ls -1
    - |
      bash cicd/build-changes.sh
  tags:
    - angular
```

5. Optimización de la construcción de imágenes Docker

5.1. Ajuste de variables

Al haber creado tanto librerías y subaplicaciones para los módulos de la aplicación, tenemos que ajustar las variables como se observa en la Figura 3.24 y usarlos de referencia para el proceso de construcción de imágenes.

Figura 3.24: Nuevas variables para la construcción de imágenes

```
variables:
REPO: gitlab.smartcitiesperu.com
GROUP: mac-salud
PROJECT: macsalud-app-v2
PROJECT_ERP: erp-app-v2
PROJECT_SCRUM: erp-app-scrum
PROJECT_ADMISSION: erp-app-admission
PROJECT_DRUGSTORE: erp-app-drugstore
PROJECT_INSURE: erp-app-insure
PROJECT_HUMAN_RESOURCES: erp-app-human-resources

stages:
- version
- version-publish
- build
- upload-image
- deploy-chart
```

5.2. Construcción secuencial de imágenes

Para la construcción de imágenes se realizarán los siguientes pasos descritos a continuación así como también el código fuente de este proceso que veremos en la figura 3.25.

- **Entorno de distribución:** Eliminamos el enlace simbólico creado del paso anterior y copiamos recursivamente los módulos minificados a una única carpeta `dist`.
- **Construcción de imágenes Docker:** Utilizando las rutas de cada subaplicación, el script compila las imágenes Docker mediante el archivo Dockerfile correspondiente de cada subaplicación. Cada imagen se etiqueta con el nombre del proyecto y la versión actual (definida en la variable `$VERSION`), lo cual garantiza que cada imagen refleje el estado específico de la aplicación en ese momento.
- **Publicación en el registro privado:** Cada una se etiqueta para el repositorio privado registry.scp.smartc.pe:5005. Posteriormente, se realiza el push al registro, donde cada subaplicación tendrá su versión independiente, facilitando el despliegue y gestión de versiones.

Figura 3.25: Nuevo proceso de construcción secuencial de imágenes

```
upload-erp-dev:
  stage: docker-build
  only:
    - develop
  script:
    - ln -s ~/cache-macsalud-v2/dist-stg dist
    - rm dist
    - cp -R ~/cache-macsalud-v2/dist-stg dist

    - docker build -t $PROJECT:$VERSION -f cicd/docker/Dockerfile .
    - docker tag $PROJECT:$VERSION registry.scp.smartc.pe:5005/$PROJECT:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT:$VERSION
    - docker build -t $PROJECT_SCRUM:$VERSION -f projects/erp-app-scrum/cicd/docker/Dockerfile .
    - docker tag $PROJECT_SCRUM:$VERSION registry.scp.smartc.pe:5005/$PROJECT_SCRUM:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT_SCRUM:$VERSION
    - docker build -t $PROJECT_ADMISSION:$VERSION -f projects/erp-app-admission/cicd/docker/Dockerfile .
    - docker tag $PROJECT_ADMISSION:$VERSION registry.scp.smartc.pe:5005/$PROJECT_ADMISSION:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT_ADMISSION:$VERSION
    - docker build -t $PROJECT_DRUGSTORE:$VERSION -f projects/erp-app-drugstore/cicd/docker/Dockerfile .
    - docker tag $PROJECT_DRUGSTORE:$VERSION registry.scp.smartc.pe:5005/$PROJECT_DRUGSTORE:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT_DRUGSTORE:$VERSION
    - docker buld -t $PROJECT_INSURE:$VERSION -f projects/erp-app-insure/cicd/docker/Dockerfile .
    - docker tag $PROJECT_INSURE:$VERSION registry.scp.smartc.pe:5005/$PROJECT_INSURE:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT_INSURE:$VERSION
    - docker build -t $PROJECT_HUMAN_RESOURCES:$VERSION -f projects/erp-app-human-resources/cicd/docker/Dockerfile .
    - docker tag $PROJECT_HUMAN_RESOURCES:$VERSION registry.scp.smartc.pe:5005/$PROJECT_HUMAN_RESOURCES:$VERSION
    - docker push registry.scp.smartc.pe:5005/$PROJECT_HUMAN_RESOURCES:$VERSION
  tags:
    - general
```

5.3. Construcción paralela de imágenes

Para optimizar el proceso descrito en la figura 3.25, construiremos las imágenes contenedorizadas en paralelo, se ha tomado en cuenta que tanto la aplicación principal como las cuatro subaplicaciones ya cuentan con archivos minificados. Esto significa que solo es necesario construir las imágenes, ya que ninguna de ellas depende de las demás. Gracias a esto, es posible realizar la construcción paralela de imágenes utilizando `parallel - matrix` en el archivo .yaml del pipeline. Para que esta paralelización funcione correctamente, es necesario confirmar que el archivo config.toml del runner ubicado generalmente en `/etc/gitlab-runner` esté configurado para ejecutar scripts en paralelo, asegurando así el máximo aprovechamiento de los recursos disponibles (ver Figura 3.26).

Figura 3.26: Configuración del runner para ejecutar trabajos en paralelo

```
concurrent = 4
check_interval = 0

[session_server]
   session_timeout = 1800

[[runners]]
   name = "scp-101"
   url = "https://gitlab.smartcitiesperu.com/"
   token = "S5DufWskrCfPE5AVpw--"
   executor = "shell"
   clone_url = "https://gitlab.smartcitiesperu.com/"
   [runners.custom_build_dir]
   [runners.cache]
      [runners.cache.s3]
      [runners.cache.gcs]
      [runners.cache.azure]
```

El step del pipeline configurado para construcciones de imágenes en paralelo se muestra a continuación en la figura 3.27.

Figura 3.27: Pipeline para ejecutar trabajos en paralelo

```yaml
build-and-push:
  stage: docker-build
  only:
    - develop
  parallel:
    matrix:
      - IMAGE_NAME: $PROJECT
        CUSTOM_PATH: 'dist'
        DOCKERFILE_PATH: cicd/docker/Dockerfile
      - IMAGE_NAME: $PROJECT_SCRUM
        DOCKERFILE_PATH: projects/erp-app-scrum/cicd/docker/Dockerfile
      - IMAGE_NAME: $PROJECT_ADMISSION
        DOCKERFILE_PATH: projects/erp-app-admission/cicd/docker/Dockerfile
      - IMAGE_NAME: $PROJECT_DRUGSTORE
        DOCKERFILE_PATH: projects/erp-app-drugstore/cicd/docker/Dockerfile
      - IMAGE_NAME: $PROJECT_INSURE
        DOCKERFILE_PATH: projects/erp-app-insure/cicd/docker/Dockerfile
      - IMAGE_NAME: $PROJECT_HUMAN_RESOURCES
        DOCKERFILE_PATH: projects/erp-app-human-resources/cicd/docker/Dockerfile
  script:
    - echo "IMAGE_NAME $IMAGE_NAME"
    - echo "VERSION $VERSION"
    - echo "DOCKERFILE_PATH $DOCKERFILE_PATH"
    - echo "CUSTOM_PATH $CUSTOM_PATH"
    - |
      if [ -n "$CUSTOM_PATH" ]; then
        echo "Copy all dist"
         cp -R ~/cache-macsalud-v2/dist-stg dist
      else
        echo "Copy only folder"
        mkdir -p dist/$IMAGE_NAME
        cp -R ~/cache-macsalud-v2/dist-stg/$IMAGE_NAME/* dist/$IMAGE_NAME/
      fi
    - ls dist
    - docker build -t "$IMAGE_NAME:$VERSION" -f "$DOCKERFILE_PATH" .
    - docker tag "$IMAGE_NAME:$VERSION" "registry.scp.smartc.pe:5005/$IMAGE_NAME:$VERSION"
    - docker push "registry.scp.smartc.pe:5005/$IMAGE_NAME:$VERSION"
    - rm -rf dist/*
  dependencies:
    - version
    - build-dev
  tags:
    - angular
```

6. Metodología del despliegue para múltiples repositorios

Se migró desde un modelo de despliegue directo en los pipelines de cada repositorio a uno centralizado y escalable con pipelines encadenados. Ahora, un pipeline principal desencadena otro secundario que consulta la última versión del artefacto desde la API de GitLab, actualiza las configuraciones en los charts de Helm con yq y realiza el despliegue en Kubernetes, este proceso se evidencia en la figura 3.28.

Figura 3.28: Obtención de la ultima versión generada y actualizar los values de Heml

```
gen-version:
  stage: version
  only:
    - develop
  script:
    - >
      APP_V2_VERSION=$(curl -Ss --request GET --header "PRIVATE-TOKEN:kM4w6oiPxYmzhvkGpH9-"
"https://gitlab.smartcitiesperu.com/api/v4/projects/142/repository/tags?sort=desc&order_by=updated" | jq -r '.[0] |
.name')
    - echo "APP_V2_VERSION=$APP_V2_VERSION" >> version_values.env
    - cat version_values.env
    - >
      exe_command="
        yq -i '.appv2.image.tag = strenv(APP_V2_VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
        yq -i '.appAdmission.image.tag = strenv(APP_V2_VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
        yq -i '.appDrugstore.image.tag = strenv(APP_V2_VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
        yq -i '.appInsure.image.tag = strenv(APP_V2_VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
        yq -i '.appHumanResources.image.tag = strenv(APP_V2_VERSION)' ~/macsalud-devops/dev/chart/erp-dev/values.yaml;
      "
    - ssh -t scpadm@192.168.71.200 -p 22 "$exe_command"
  tags:
    - general
  artifacts:
    paths:
      - version_values.env
```

En esta parte, replicamos el proceso de despliegue existente, asegurando que Kubernetes, en conjunto con Helm, gestione la actualización de las aplicaciones mediante las nuevas versiones de las imágenes creadas. Este procedimiento incluye la limpieza de recursos antiguos y la implementación de las configuraciones actualizadas, lo que garantiza un despliegue consistente y confiable (como se evidencia en la Figura 3.29).

Figura 3.29: Ejecución del despliegue de las nuevas versiones

```
deploy-dev:
  stage: deploy
  only:
    - develop
  script:
    - cat version_values.env
    - source version_values.env
    - >
      exe_command="
        kubectl --kubeconfig=/home/scpadm/.kube/config-local --namespace=erp-dev delete ingress erp-dev-ingress;
        cd ~/macsalud-devops/dev/chart;
        helm upgrade erp-dev ./erp-dev --namespace=erp-dev --kubeconfig=/home/scpadm/.kube/config-local;
      "
    - ssh -t scpadm@192.168.71.200 -p 22 "$exe_command"
  tags:
    - general
```

6.1. Pipeline optimizado

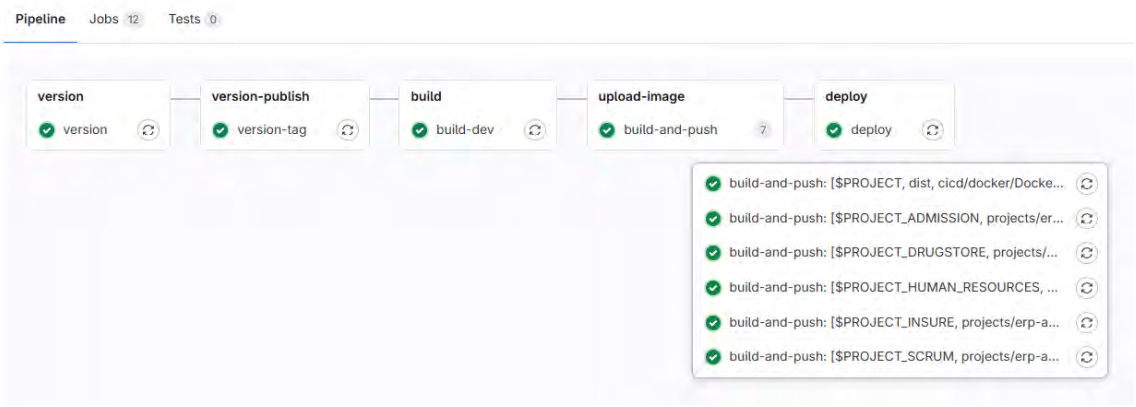
En la Figura 3.30 se observa la estructura del pipeline posterior a la optimización implementada en GitLab CI/CD. A diferencia del pipeline inicial, en este modelo se incorporaron mejoras clave orientadas a reducir los tiempos de compilación y despliegue, así como a optimizar el uso de recursos. El pipeline optimizado contempla las siguientes etapas:

- **Version:** generación de la versión del sistema.
- **Version-publish:** publicación de la nueva versión en el repositorio correspondiente.

- **Build:** compilación selectiva de librerías y subaplicaciones, ejecutada a través del script `build-changes.sh`, que permite identificar únicamente los módulos modificados. Esta etapa constituye una de las mejoras más significativas respecto al pipeline anterior, ya que reduce de manera considerable los tiempos de compilación de los módulos asignados.
- **Upload-image:** construcción y subida en paralelo de múltiples imágenes Docker a un registro centralizado. También esta etapa reduce el tiempo de construcción total ya que logra distribuir la carga de trabajo.
- **Deploy:** despliegue automatizado de los módulos actualizados en el entorno de ejecución, gestionado mediante orquestadores y garantizando independencia entre subaplicaciones.

Este nuevo pipeline permite que los cambios aplicados a un módulo específico no obliguen a reconstruir ni desplegar toda la aplicación, sino únicamente los componentes afectados.

Figura 3.30: Pipeline optimizado en GitLab CI/CD tras aplicar el modelo propuesto



7. Instalación básica de Helm

En esta sección se describen los pasos necesarios para instalar Helm en un entorno Kubernetes, una herramienta clave para la gestión de aplicaciones y despliegues en contenedores. Aunque en la empresa "Smart Cities Perú" ya se estaba utilizando Helm, este proceso es útil para aquellos que implementan Helm en entornos nuevos o en sistemas que aún no están configurados con esta herramienta.

Dado que la mayoría de los servidores y entornos de producción para aplicaciones basadas en Kubernetes utilizan sistemas operativos Linux, se recomienda realizar la instalación de Helm en este tipo de entornos. Esta opción asegura una mayor compatibilidad y facilidad en la gestión de herramientas como Kubernetes y Docker, que son comúnmente desplegadas en servidores Linux.

En la figura 3.31 se detallan los pasos básicos de instalación de Helm. No se detallan de forma específica, ya que no es el objetivo principal de esta tesis.

Figura 3.31: Proceso de instalación básica de Helm

```
# Actualiza tu lista de paquetes:
- sudo apt-get update
#Instala las dependencias necesarias:
- sudo apt-get install curl
# Descarga y extrae el archivo de Helm:
- |
    curl https://get.helm.sh/helm-v3.10.0-linux-amd64.tar.gz --output helm-v3.10.0-linux-amd64.tar.gz
    tar -zxvf helm-v3.10.0-linux-amd64.tar.gz
# Mueve el binario de Helm a tu PATH:
- sudo mv linux-amd64/helm /usr/local/bin/helm
# Verifica la instalación de heml
- helm version
```

Luego se procede a crear un "Chart" de Heml, a continuación se presentan los pasos que se deben de realizar (ver Figura 3.32).

Figura 3.32: Proceso de creación de un chart en Helm

```
# Actualiza tu lista de paquetes:
- helm create my-chart
# Instala el chart en tu cluster de Kubernetes
- helm install my-release ./my-chart
# Verifica el despliegue
- kubectl get all
```

8. Configuración de redirecciones de Ingress para despliegue con Kubernetes

En cuanto a la configuración de las redirecciones, se utiliza un recurso de tipo Ingress en Kubernetes, que permite manejar las rutas hacia los distintos módulos de la aplicación ya que separamos en subaplicaciones los módulos (admission, drugstore, insure, human-resources). En el archivo de configuración Ingress (controlador de acceso externo para servicios en Kubernetes), se especifican reglas que redirigen el tráfico según el prefijo de la URL, como /admission y /drugstore, hacia los servicios correspondientes de cada módulo en el clúster. Este Ingress está configurado para gestionar el tamaño máximo del cuerpo, así como los tiempos de conexión y respuesta, asegurando la estabilidad en la transferencia de datos. La configuración de dichas redirecciones de visualiza en la Figura 3.33.

Figura 3.33: Configuración de Ingress para las redirecciones a las sub-aplicaciones

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ include "erp-dev.fullname" . }}-ingress
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "100m"
    nginx.ingress.kubernetes.io/proxy-connect-timeout: '600'
    nginx.ingress.kubernetes.io/proxy-send-timeout: '600'
    nginx.ingress.kubernetes.io/proxy-read-timeout: '600'
spec:
  ingressClassName: {{ .Values.ingress.className }}
  rules:
    - host: macsalud.dev.erp.onscp.com
      http:
        paths:
          - path: /admission
            pathType: Prefix
            backend:
              service:
                name: {{ include "erp-dev.name" . }}-app-admission-service
                port:
                  number: 8080
          - path: /drugstore
            pathType: Prefix
            backend:
              service:
                name: {{ include "erp-dev.name" . }}-app-drugstore-service
                port:
                  number: 8080
          - path: /
            pathType: Prefix
            backend:
              service:
                name: {{ include "erp-dev.name" . }}-app-v2-service
                port:
                  number: 8085
           pathType: Prefix
```

8.1. Configuración de los manifiestos de Service y Deployment

A continuación, en las Figuras 3.34 y 3.35 se muestran las configuraciones de los manifiestos Service y Deployment, aplicada a la redirección de la ruta /admission. Esta estructura se podrá reutilizar en las configuraciones para las demás subaplicaciones.

Figura 3.34: Deployment para la subaplicación admisión

```
metadata:
  name: {{ include "erp-dev.fullname" . }}-app-admission
spec:
  selector:
    matchLabels:
      app: {{ include "erp-dev.name" . }}-app-admission
  replicas: 1
  template:
    metadata:
      labels:
        app: {{ include "erp-dev.name" . }}-app-admission
    spec:
      containers:
        - name: erp-app-admission
          image: {{ .Values.appAdmission.image.repository }}:{{ .Values.appAdmission.image.tag
}}        ports:
            - containerPort: 80
          env:
            - name: NAMESPACE
              value: "stg2"
            - name: LOGO_URL
              valueFrom:
                secretKeyRef:
                  name: {{ include "erp-dev.fullname" . }}-secrets
                  key: LOGO_URL
```

Figura 3.35: Service usado para la subaplicacion admisión

```
kind: Service
apiVersion: v1
metadata:
  name: {{ include "erp-dev.name" . }}-app-admission-service
spec:
  selector:
    app: {{ include "erp-dev.name" . }}-app-admission
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
  type: NodePort
```

8.2. Uso de los charts de Helm para manejar los valores de configuración

Finalmente, para el despliegue de las imágenes Docker de cada módulo en Kubernetes, se utiliza Helm con sus charts (paquete reutilizable que contiene todo lo necesario para desplegar una aplicación en kubernetes), lo que facilita la gestión centralizada de valores de configuración, el control de versiones y la automatización de los despliegues en distintos entornos.

Las versiones mostradas en la Figura 3.36 corresponden a versiones de imágenes contenedorizadas que deben estar disponibles en el registro de imágenes, las cuales fueron generados durante el proceso de construcción docker de la aplicación ERP, descrito previamente en la etapa de CI/CD.

Figura 3.36: Valores de configuración usados en las plantillas de los charts

```
appv2:
  image:
    repository: "registry.scp.smartc.pe:5005/macsalud-app-v2"
    tag: "v1.458.5"
appScrum:
  image:
    repository: "registry.scp.smartc.pe:5005/erp-app-scrum"
    tag: "v1.458.5"
appAdmission:
  image:
    repository: "registry.scp.smartc.pe:5005/erp-app-admission"
    tag: "v1.458.5"
appDrugstore:
  image:
    repository: "registry.scp.smartc.pe:5005/erp-app-drugstore"
    tag: "v1.458.5"
appInsure:
  image:
    repository: "registry.scp.smartc.pe:5005/erp-app-insure"
    tag: "v1.458.5"
appHumanResources:
  image:
    repository: "registry.scp.smartc.pe:5005/erp-app-human-resources"
    tag: "v1.458.5"
```

Capítulo 4

Benchmarking y pruebas de rendimiento

1. Optimización de tiempos de compilación

1.1. Proceso de compilación antes de la optimización

La Tabla 4.1 muestra los tiempos de ejecución total del pipeline antes de realizar optimizaciónes, que incluye el proceso tradicional de compilación de la aplicación completa mediante "ng build", su posterior creación de la imagen docker con el comando "docker build" y el proceso de despliegue con Helm.

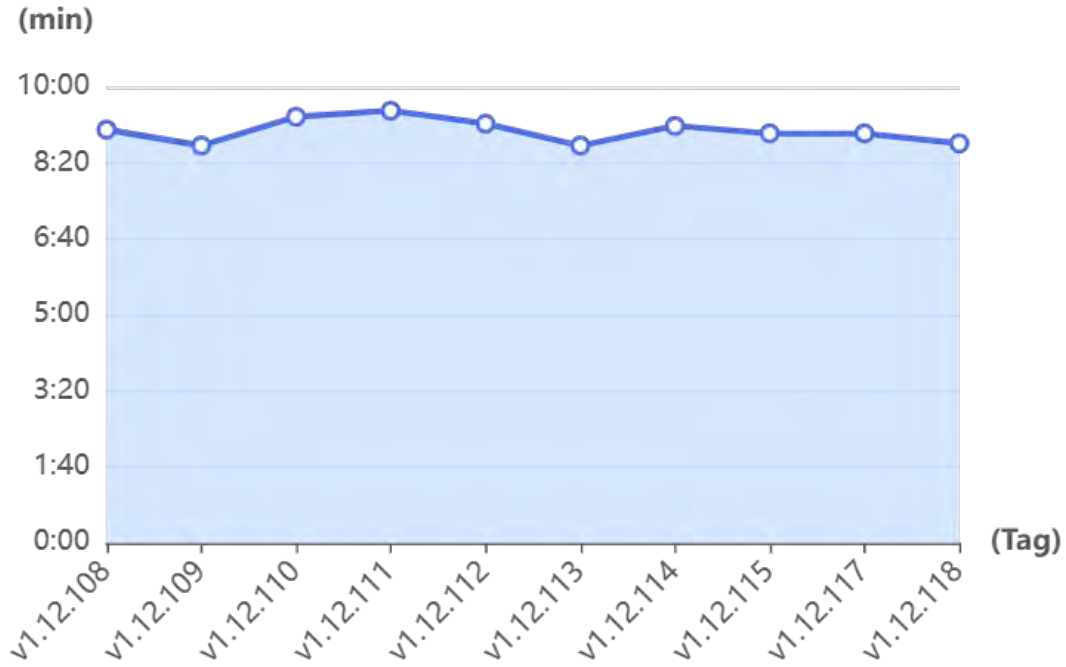
Tabla 4.1: Tabla de tiempos de ejecución del pipeline antes de la optimización

Tag	**Proyecto**	**Módulos**	**Duración Total (minutos)**
v1.12.108	ERP	rrhh	9:04
v1.12.109	ERP	admisión, seguros	8:43
v1.12.110	ERP	scrum, tesorería	9:21
v1.12.111	ERP	shared	9:29
v1.12.112	ERP	farmacia	9:12
v1.12.113	ERP	seguros	8:43
v1.12.114	ERP	tesorería, logística	9:09
v1.12.115	ERP	admisión	8:59
v1.12.117	ERP	seguros	8:59
v1.12.118	ERP	farmacia	8:46

Los tiempos obtenidos de la ejecución del pipeline (compilación y despliegue), independientemente del módulo del ERP que haya sido modificado, el sistema completo es recompilado y reconstruido en cada despliegue. Esto significa que incluso cambios aislados en módulos como rrhh, farmacia o shared generan una recompilación completa de la aplicación principal.

La Figura 4.1 muestra el comportamiento de estos tiempos, evidenciando una duración consistente cercana a los 9 minutos por compilación, independientemente del módulo modificado, lo cual representa una demora considerable, especialmente cuando se requiere aplicar correcciones urgentes en producción.

Figura 4.1: Tiempos de ejecución del pipeline antes de la optimización



1.2. Compilación y contenedorización modular

Esta sección presenta los tiempos de ejecución de los jobs correspondientes al proceso de compilación y construcción de librerías y subaplicaciones, obtenidos tras modularizar el ERP.

Compilación y contrucción de librerías y subaplicaciones

La Tabla 4.2 muestra datos obtenidos del proceso de compilación, construcción secuencial de las imágenes docker y el tiempo total que tomo completar todo el pipeline para cambios en librerías y subaplicaciones del ERP. También se muestra la Figura 4.2 donde se muestra el comportamiento de la compilación de los módulos a lo largo de varios despliegues así también como los tiempos de construcción de las imágenes docker en la Figura 4.3.

Tabla 4.2: Tabla de tiempos de ejecución del pipeline después de la primera optimización

Tag	Módulos	Compilación (min)	Construcción (seg)	Total
v1.12.89	Mant. SG, Admisión	4:55	31	5:33
v1.12.93	Archivo, Seguros	4:12	43	5:03
v1.12.94	Core, Admisión	4:51	35	5:33
v1.12.98	Core, Admisión	4:43	29	5:19
v1.12.105	Gerencia, Admisión	4:32	29	5:08
v1.12.116	Archivo, Seguros	4:02	37	4:46
v1.12.117	Archivo, Farmacia	4:23	29	4:59
v1.12.119	RRHH lib, RRHH app	5:07	30	5:44
v1.12.120	Tesorería, Farmacia	4:46	35	5:28
v1.12.122	RRHH lib, RRHH app	5:08	28	5:43

Figura 4.2: Tiempos de compilación (lib y subapps) basado en la primera optimización

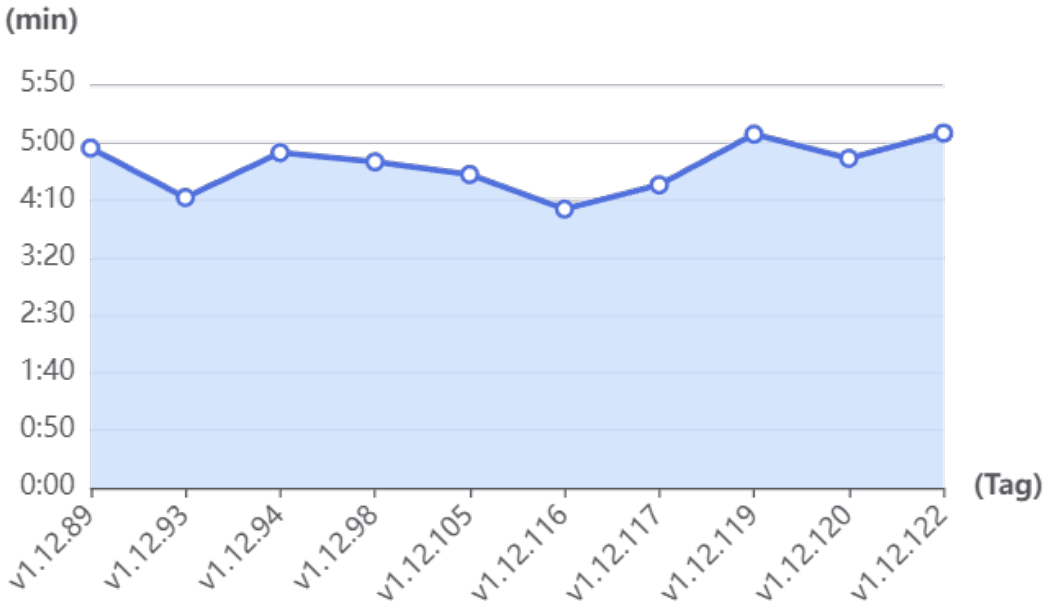
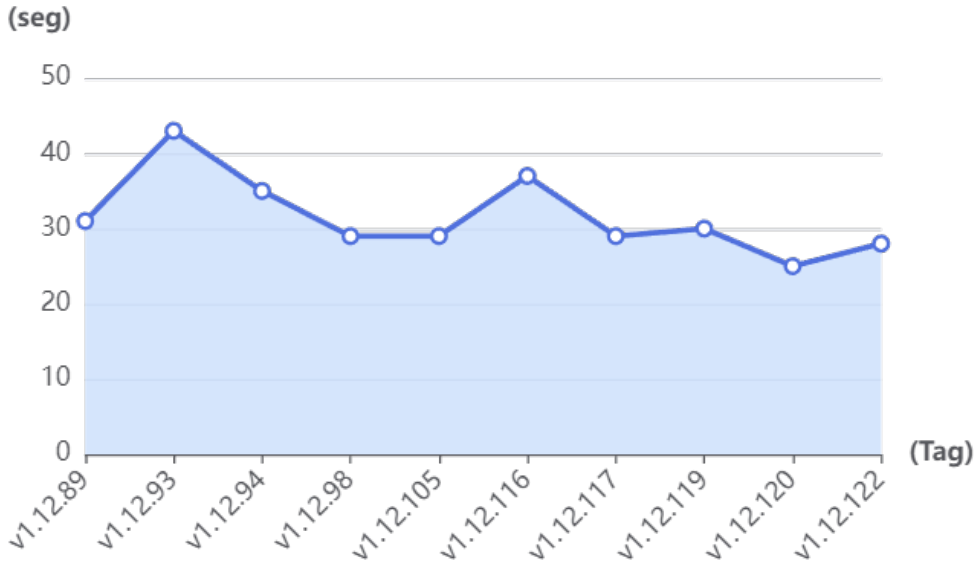
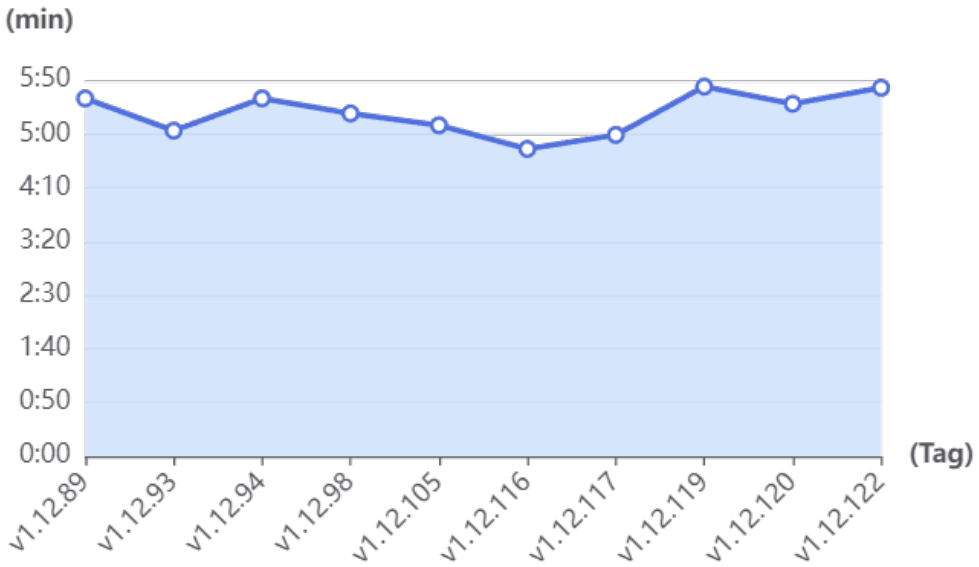


Figura 4.3: Tiempos de construcción basado en la primera optimización



La Figura 4.4 muestra resultados del tiempo total de ejecución del pipeline en esta etapa después de realizar la construcción tanto de la aplicación principal (que empaqueta a las librerías y archivos minificados) y las subaplicaciones generadas del ERP.

Figura 4.4: Tiempos de ejecución de pipelines basado en la primera optimización



Compilación y construcción de solo subaplicaciones

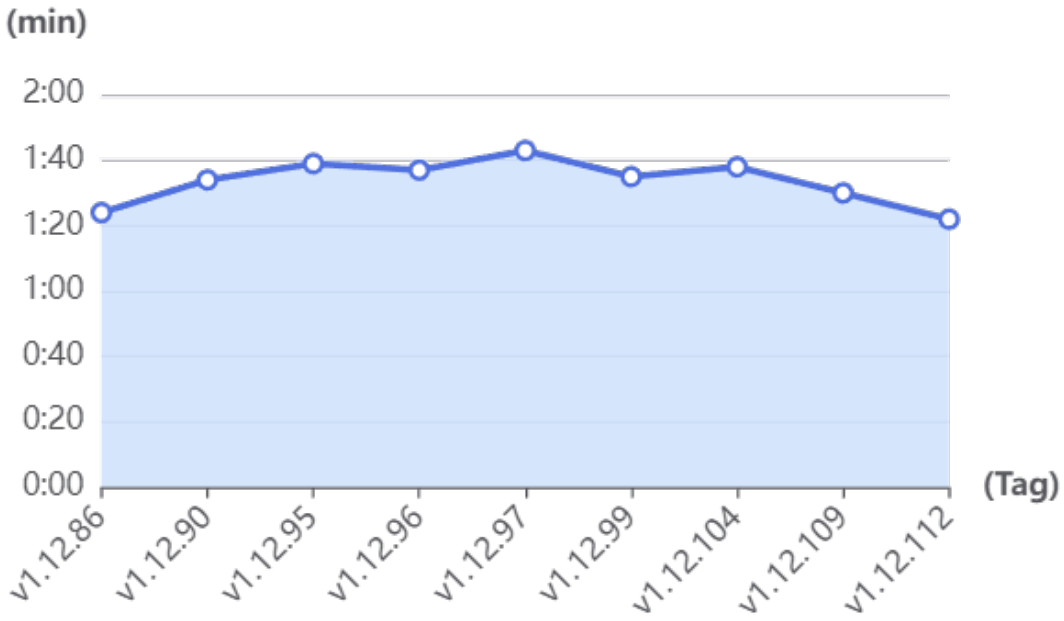
La Tabla 4.3 presenta los datos obtenidos durante el proceso de compilación, la construcción secuencial de las imágenes Docker y el tiempo total requerido para completar la ejecución del pipeline. Estos resultados corresponden a escenarios en los que los cambios se realizaron únicamente en las subaplicaciones generadas.

Tabla 4.3: Tabla de tiempos de ejecución del pipeline después de la primera optimización (subaplicaciones)

Tag	Módulos	Compilación (min)	Construcción (seg)	Total
v1.12.86	Seguros	1:24	43	2:14
v1.12.90	Farmacia	1:34	32	2:13
v1.12.95	Admisión	1:39	31	2:17
v1.12.96	Admisión	1:37	35	2:20
v1.12.97	Admisión	1:43	31	2:21
v1.12.99	Farmacia	1:35	32	2:14
v1.12.104	Farmacia	1:38	29	2:13
v1.12.109	Farmacia	1:30	32	2:09
v1.12.112	Seguros	1:22	30	1:58

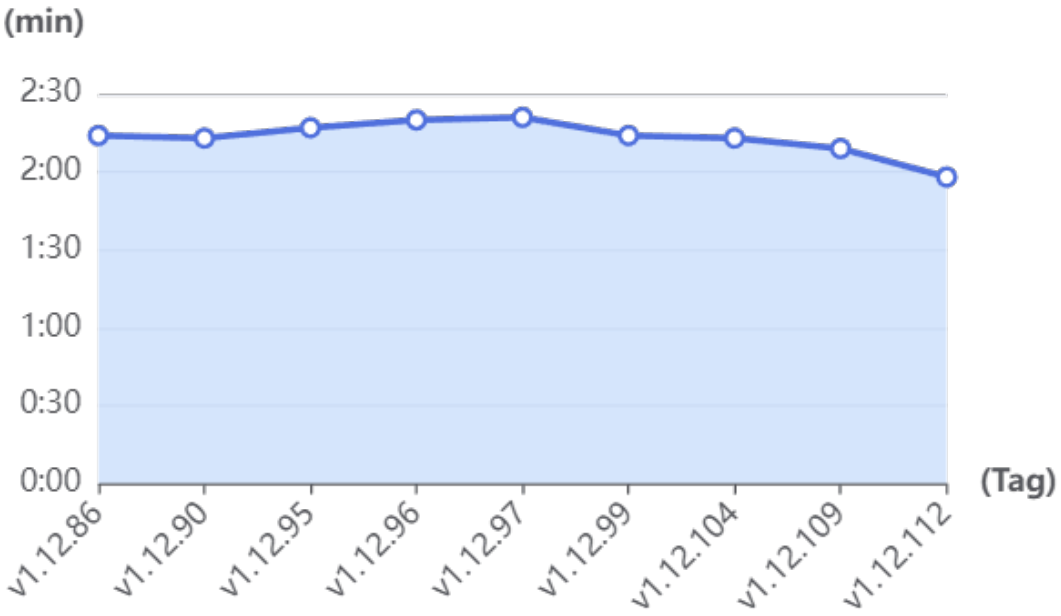
La Figura 4.5 representa el comportamiento de los tiempos asociados a la compilación de subaplicaciones generadas en base a cambios realizados en el ERP a lo largo de varios despliegues hechos.

Figura 4.5: Tiempos de compilación (subapps) basado en la primera optimización



La Figura 4.6 representa el comportamiento de los tiempos totales de ejecución de los pipelines en base a cambios realizados en subaplicaciones del ERP a lo largo de varios despliegues realizados.

Figura 4.6: Tiempos de ejecución de pipelines basado en la primera optimización



1.3. Contenedorización paralela

A continuación, en la Tabla 4.4 presentamos los tiempos asociados a la segunda optimización el cual se basa en la ejecución simultanea de tareas en el pipeline, nos enfocaremos a la paralelización de contrucción de imagenes docker para cada subaplicacion trabajada y finalmente para la app principal.

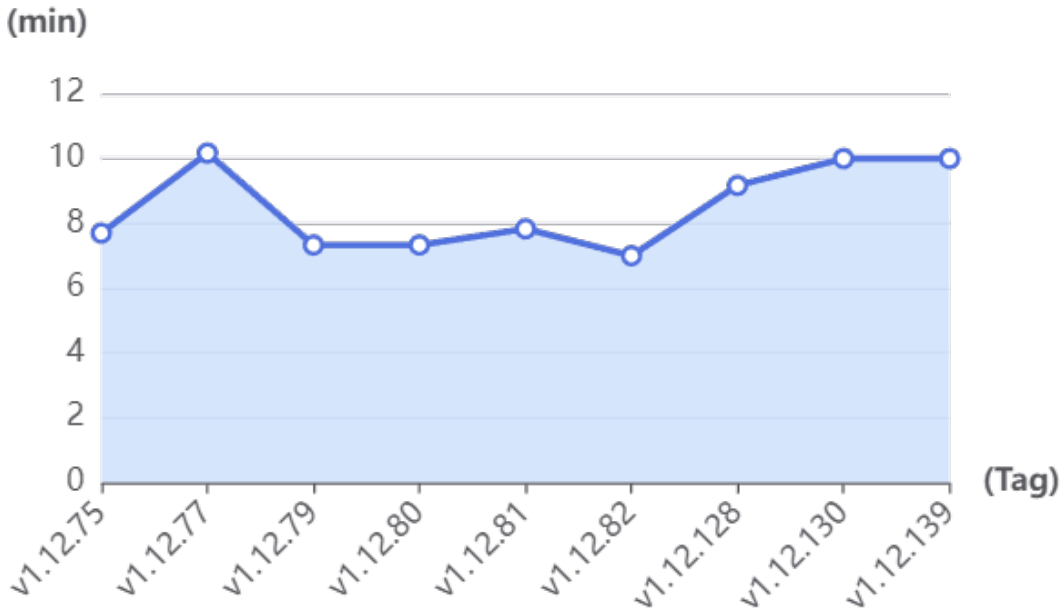
El orden de los tiempos por cada subaplicacion y aplicación principal es el siguiente: admisión, app principal, farmacia, RRHH, seguros, scrum.

Tabla 4.4: Tiempos de construcción de imágenes docker después de la segunda optimización

Tag	Tiempos por cada subaplicacion (seg)	Duración promedio (seg)
v1.12.75	8, 12, 8, 6, 5, 4	7.7
v1.12.77	12, 17, 12, 6, 6, 8	10.17
v1.12.79	8, 12, 8, 6, 5, 5	7.33
v1.12.80	8, 12, 8, 6, 6, 4	7.33
v1.12.81	9, 12, 8, 6, 6, 6	7.83
v1.12.82	8, 11, 7, 6, 6, 4	7
v1.12.128	8, 14, 12, 8, 9, 4	9.17
v1.12.130	12, 15, 12, 6, 6, 9	10
v1.12.139	10, 14, 10, 7, 9, 3	10

El comportamiento en tiempos de construcción paralela de la aplicación principal y las subaplicaciones se visualiza en la Figura 4.7 a continuación.

Figura 4.7: Tiempos de construcción de imágenes docker en paralelo



1.4. Comparación antes y después de aplicar optimizaciones en CI/CD

Pipelines con cambios solo en subaplicaciones

Las Figuras 4.8, 4.9, 4.10, 4.11, 4.12 y 4.13 muestran los pipelines ejecutados antes y después de aplicar los cambios en las subaplicaciones (admisión, seguros y farmacia). Estas figuras permiten observar cómo era el flujo inicial y cómo quedó luego de las modificaciones, facilitando la comparación entre ambos estados del proceso.

Figura 4.8: Pipeline de cambios en admisión (antes)

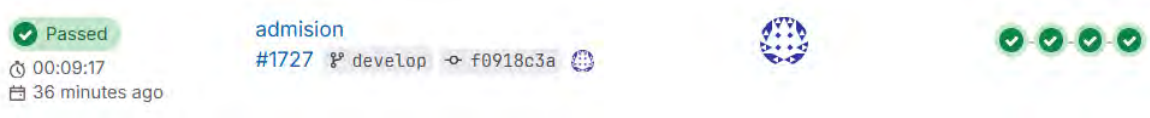


Figura 4.9: Pipeline de cambios en admisión (después)

Figura 4.10: Pipeline de cambios en seguros (antes)

Figura 4.11: Pipeline de cambios en seguros (después)

Figura 4.12: Pipeline de cambios en farmacia (antes)

Figura 4.13: Pipeline de cambios en farmacia (después)

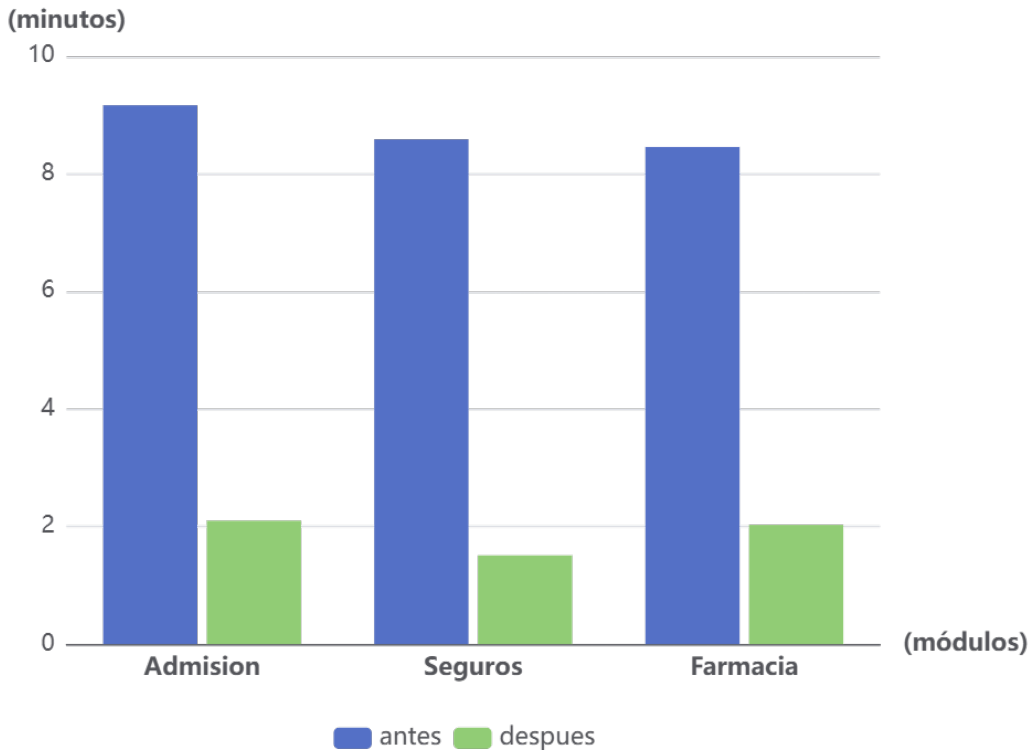
A continuación, la Tabla 4.5 presenta una comparación de los tiempos de ejecución de los pipelines antes y después de los cambios, enfocándose en aquellos módulos que actualmente sufren modificaciones de forma recurrente.

Tabla 4.5: Tabla de tiempos de ejecución de pipelines antes y después de realizar optimizaciones

Version	**Módulos**	**Antes (min)**	**Después (min)**
v1.12.116 – v1.13.7	Admision	09:17	2:10
v1.12.117 – v1.12.8	Seguros	08:59	1:51
v1.12.118 – v1.13.9	Farmacia	08:46	2:03

La Figura 4.14 presenta de forma visual los tiempos de ejecución de los pipelines antes y después de los cambios, enfocándose en las subaplicaciones más afectadas por actualizaciones constantes. Se evidencia una gran diferencia de tiempos totales de ejecución hasta el despliegue final.

Figura 4.14: Tiempos de ejecución de los pipelines antes y después de la optimización para cambios en subaplicaciones



Pipelines con cambios solo en librerías y subaplicaciones

Las Figuras 4.15, 4.16, 4.17, 4.18, 4.19 y 4.20 muestran los pipelines ejecutados antes y después de aplicar cambios en una librería (core, tesorería, logística) y una la subaplicación (admisión, seguros, farmacia). Estas figuras permiten observar cómo era el flujo inicial y cómo quedó luego de modificaciones en 2 módulos, facilitando la comparación entre ambos estados del proceso.

Figura 4.15: Pipeline de cambios en core y admisión (antes)

Figura 4.16: Pipeline de cambios en core y admisión (después)

Figura 4.17: Pipeline de cambios en tesorería y seguros (antes)

Figura 4.18: Pipeline de cambios en tesorería y seguros (después)

Figura 4.19: Pipeline de cambios en logística y farmacia (antes)

Figura 4.20: Pipeline de cambios en logística y farmacia (después)

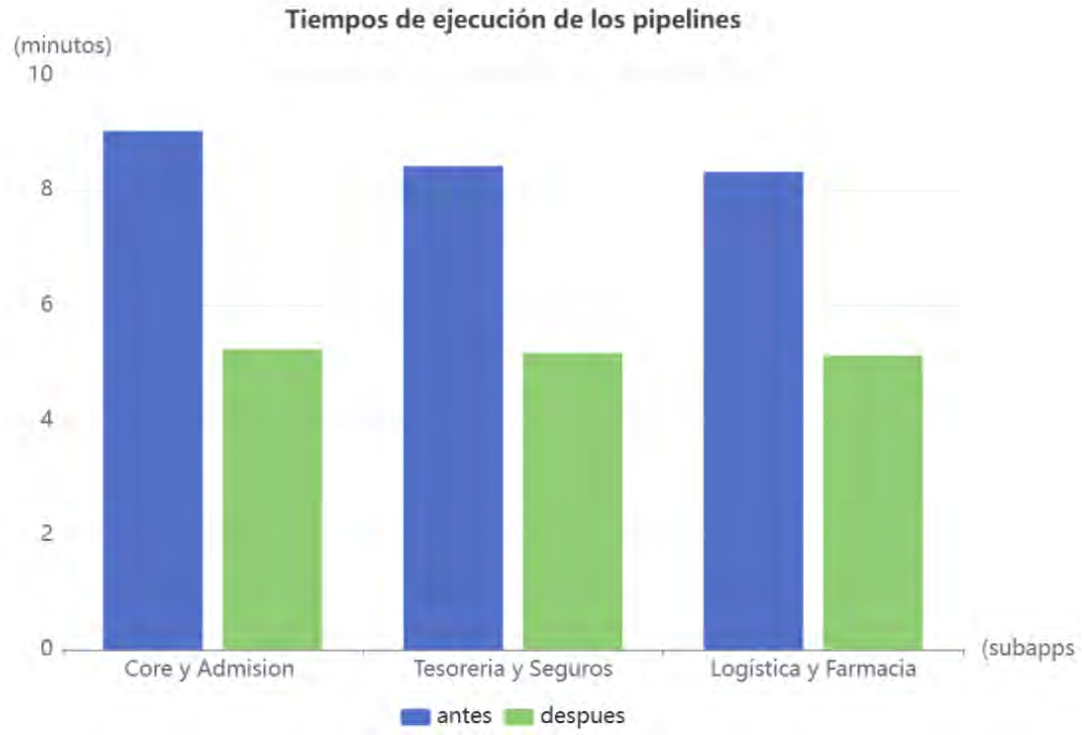
La Tabla 4.6 muestra los tiempos totales requeridos al subir cambios en una librería y en una subaplicación, antes y después de las optimizaciones.

Tabla 4.6: Tabla de tiempos de ejecución de pipelines antes y después de realizar optimizaciones

Version	**Módulos**	**Antes (min)**	**Después (min)**
v1.12.119 – v1.13.10	Core y Admisión	09:04	05:24
v1.12.120 – v1.13.11	Tesorería y Seguros	08:43	05:17
v1.12.121 – v1.13.12	Logística y Farmacia	08:33	05:13

La Figura 4.21 presenta una comparación visual de los tiempos de ejecución de los pipelines antes y después de los cambios aplicados, tanto para librerías como para subaplicaciones. Se puede observar una mejora significativa en los tiempos de ejecución, lo que refleja una optimización efectiva en el proceso de integración y despliegue continuo. Esta reducción impacta positivamente en la eficiencia del equipo de desarrollo, permitiendo ciclos de entrega más rápidos y frecuentes.

Figura 4.21: Tiempos de ejecución de los pipelines antes y después para librerías y subaplicaciones



1.5. Gráfico final de tiempos de ejecución de los pipelines antes y después

A continuación en las Figuras 4.22 y 4.23 muestran de manera grafica en base a fechas como se comportan los pipelines antes y después de haber aplicado el modelo de optimización en el Gitlab junto al runner local de la empresa estudiada.

Figura 4.22: Tiempos finales de ejecución antes de la optimización

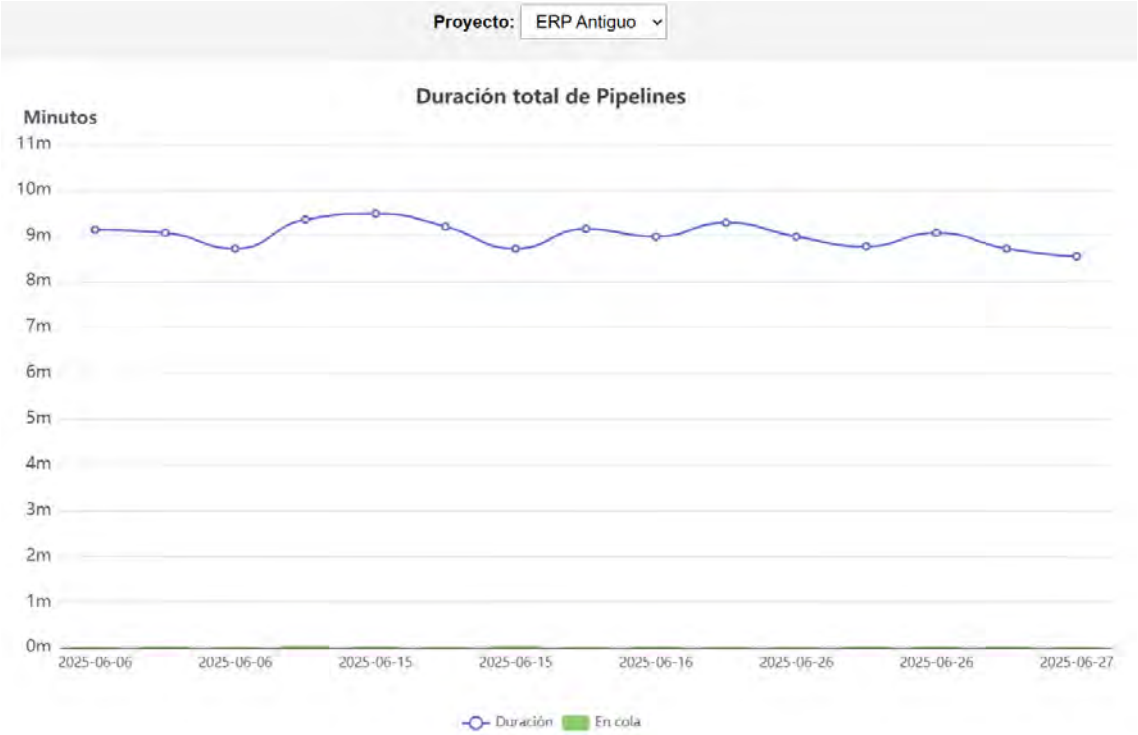
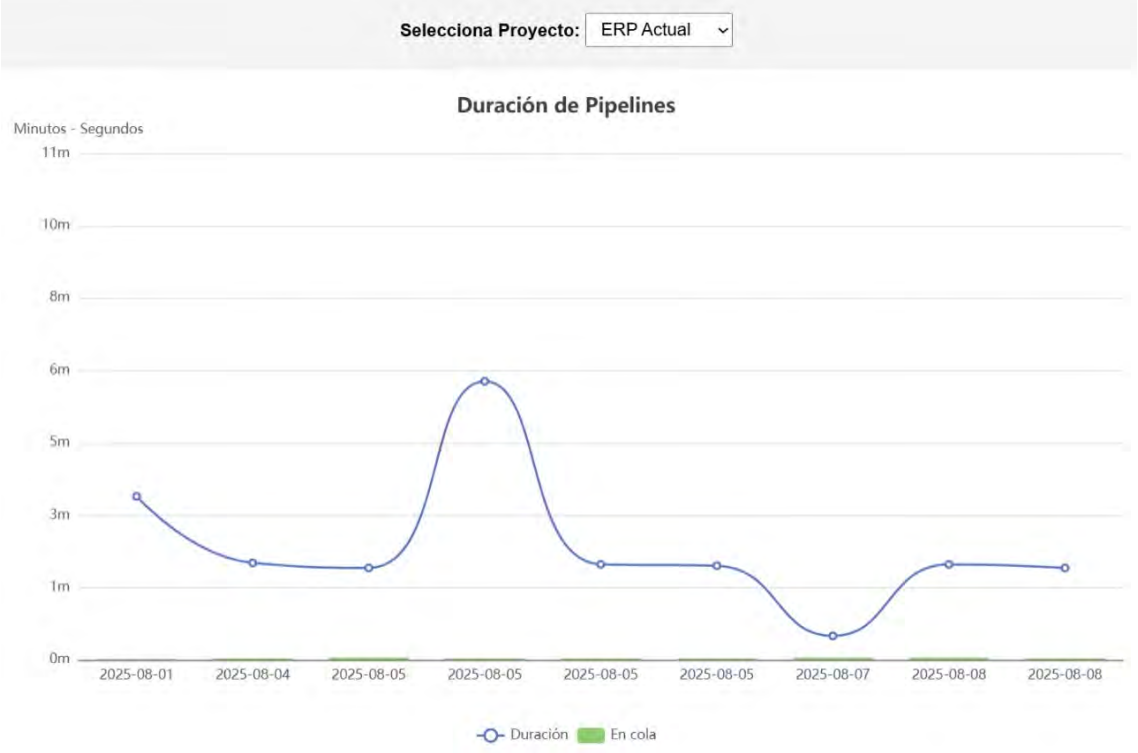


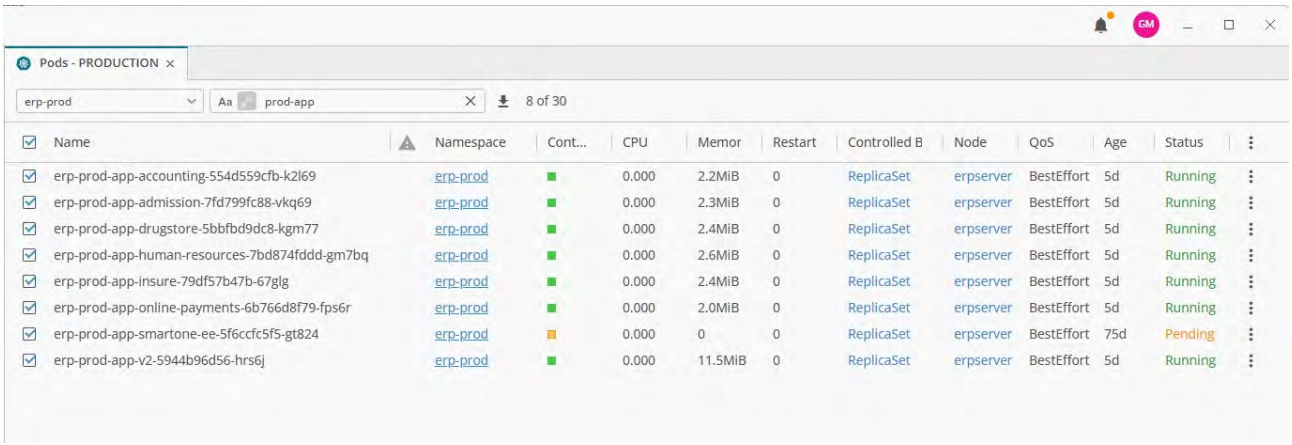
Figura 4.23: Tiempos finales de ejecución después de la optimización



2. Visualización de las subaplicaciones ejecutándose de forma independiente en producción

Al desplegar las cuatro subaplicaciones de manera independiente, es posible observar en Lens —una herramienta visual para la gestión de clústeres Kubernetes— que cada pod se ejecuta por separado. Esto permite evidenciar claramente los cambios aplicados en cada módulo, lo cual facilita el monitoreo, la validación del despliegue y la trazabilidad de los componentes en tiempo real, como se ve en la Figura 4.24.

Figura 4.24: Subaplicaciones generadas visualizadas desde Lens



3. Ahorro en costos operativos

También al aplicar estas optimizaciones y mejoras logramos 2 principales mejoras que justifican la aplicablidad de este enfoque.

3.1. Menor uso de recursos

Al compilar solo los módulos modificados y optimizar el pipeline que conlleva a ejecutarse en menor tiempo los trabajos del pipeline implica el menor uso de CPU, memoria lo que disminuye la carga en la infraestructura de CI/CD y servidores.

3.2. Eficiencia y reducción de costos en la infraestructura en la nube

La modularización y la optimización del pipeline permiten escalar de manera eficiente solo los módulos necesarios, lo que reduce costos de infraestructura en la nube (por ejemplo, AWS, Google Cloud, Azure) o servidores locales.

Aunque la empresa “Smart Cities Perú” gestiona su servidor de manera local, podemos simular el ahorro de costos en un entorno aún más profesional utilizando uno de los servicios en la nube más populares a nivel mundial, como es AWS y sus instancias de cómputo.

Instancias bajo demanda en AWS EC2

Las instancias bajo demanda son máquinas virtuales que puedes lanzar en AWS sin necesidad de hacer compromisos a largo plazo. Solo pagas por el tiempo que están activas (por segundo o por hora, dependiendo del tipo), lo cual es ideal para pruebas, compilaciones esporádicas y pipelines CI/CD que no requieren disponibilidad continua.

Importancia en la infraestructura

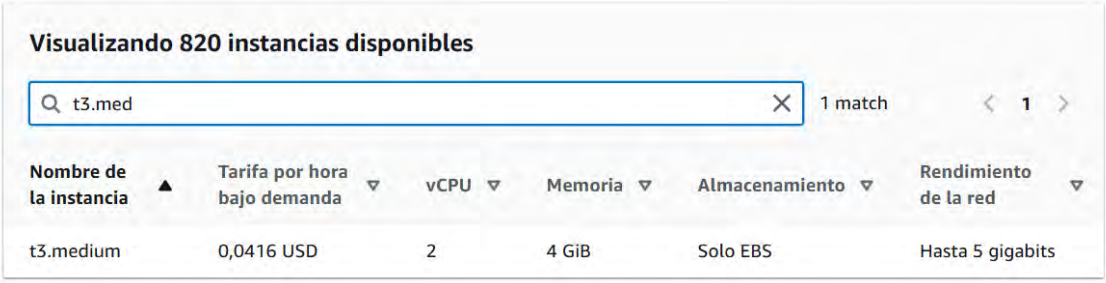
- Se compilan únicamente las librerías o subaplicaciones que han sido modificadas, lo que permite reducir significativamente el tiempo total de compilación.
- Es posible utilizar instancias EC2 bajo demanda para realizar el proceso de compilación. Estas instancias, al contar con mayor capacidad de CPU y memoria RAM, permiten acelerar el proceso. Una vez finalizada la tarea, la instancia puede ser apagada, lo que reduce costos al pagar solo por el tiempo de uso (no es necesario tener los Runner físicos encendidos).

Simulación de Ahorro en Costos de Infraestructura

En AWS, el costo de las instancias EC2 bajo demanda se calcula según el uso de capacidad de cómputo por hora, con un costo mínimo de 60 segundos. Para esta simulación, utilizamos la instancia t3.medium (instancia recomendada que soporta este tipo de trabajos), la cual tiene un costo aproximado de 0.0416 USD por hora. Esta instancia ofrece 2 vCPU, 4 GiB de memoria y un rendimiento de red de hasta 5 gigabits.

La Figura 4.25 muestra información de precios de instancias bajo demanda AWS a la fecha del 09 de julio de 2025.

Figura 4.25: Imagen de precios extraída de la pagina oficial de Amazon



Fuente: Elaboración propia

Para estimar el costo por minuto, se divide el costo por hora entre 60 minutos:

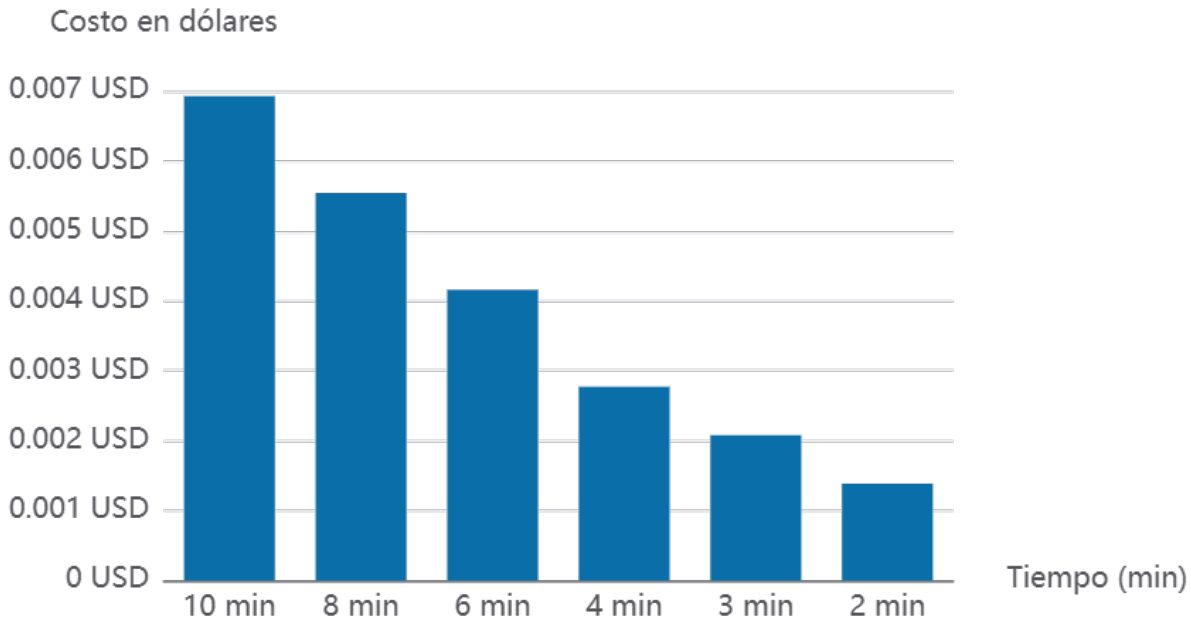
$$\text{Costo por minuto} = \frac{0.0416 \text{ USD}}{60} = 0.000693 \text{ USD por minuto}$$

Tabla 4.7: Tabla de costos por minuto en una instancia EC2 bajo demanda

Duración de un pipeline	**Costo Total**
10 min	0.00693 USD
8 min	0.005544 USD
6 min	0.004158 USD
4 min	0.002772 USD
3 min	0.002079 USD
2 min	0.001386 USD

Como se observa en la Tabla 4.7, los costos individuales por ejecución del pipeline son bastante bajos. Sin embargo, este valor corresponde únicamente a una ejecución. En escenarios mas complejos, especialmente en aplicaciones empresariales como un ERP, el pipeline puede ejecutarse con una frecuencia considerable. Por ejemplo, si se considera que el pipeline del frontend se ejecuta al menos dos veces al día, se tendría un mínimo de 60 ejecuciones al mes. Por lo tanto, optimizar la duración del pipeline tiene un impacto significativo cuando se analiza a escala mensual o en entornos con múltiples aplicaciones. Podemos ver de manera visual la tabla anterior en la Figura 4.26 presentada a continuación.

Figura 4.26: Gráfico de costos ahorrados en una instancia EC2 bajo demanda



Importancia del análisis de costos de ejecución por pipeline

Si bien el costo directo por ejecución de un pipeline puede parecer bajo cuando se analiza de forma aislada, en entornos reales este debe evaluarse considerando la frecuencia de ejecución, la cantidad de pipelines y su impacto en la productividad del equipo.

1. Modelamiento del costo en entornos CI/CD

   El costo total de ejecución puede expresarse como:

   $$C_{total} = \sum_{i=1}^{n} (E_i \times T_i \times C_m)$$

   donde E_i es el número de ejecuciones del pipeline i, T_i su duración promedio y C_m el costo por minuto.

   A partir de este modelo, es posible estimar el ahorro generado por la optimización de tiempos de ejecución.

2. Estimación del ahorro por optimización

   Considerando un escenario donde un pipeline se optimiza de 10 minutos a 3 minutos, el ahorro por ejecución es:

   $$A_{ejecucion} = 0{,}00693 - 0{,}002079 = 0{,}004851 \text{ USD}$$

   Si se tienen 3000 ejecuciones mensuales, el ahorro total sería:

   $$A_{mensual} = 3000 \times 0{,}004851 = 14{,}553 \text{ USD}$$

   Este resultado evidencia que, aunque el ahorro unitario es pequeño, su impacto acumulado es significativo.

   En escenarios con múltiples pipelines, el ahorro total puede escalar proporcionalmente. Por ejemplo, para 20 pipelines con características similares:

   $$A_{total} = 20 \times 14{,}553 = 291{,}06 \text{ USD mensuales}$$

3. Impacto en métricas de productividad

   La reducción del tiempo de ejecución impacta directamente en métricas clave como:

   - **Lead Time**: disminuye el tiempo total desde integración hasta despliegue.
   - **Throughput**: incrementa la cantidad de ejecuciones por hora.

4. Costo asociado al tiempo de espera del desarrollador

   Más allá del costo de la infraestructura, hay un costo humano: cada minuto que un desarrollador espera a que compile o despliegue su código es tiempo improductivo. Reducir de 10 a 3 minutos mejora la productividad y experiencia del desarrollador así como mayor velocidad en la entrega de valor al cliente. Este factor puede superar el costo de infraestructura, reforzando la importancia de la optimización de pipelines.

En resumen el ahorro monetario por sí solo puede parecer bajo, pero cuando se multiplica por volumen, escala y tiempo, junto con los beneficios intangibles, se convierte en una mejora estratégica, no solo técnica.

Capítulo 5

Análisis y discusión de resultados

1. Análisis de resultados respecto a los objetivos

A continuación, se presenta el análisis de los resultados obtenidos durante la implementación del modelo optimizado de CI/CD, en relación directa con los objetivos planteados y los problemas identificados al inicio del proyecto. Para ello, se consideran además los antecedentes teóricos y estudios previos que respaldan y contextualizan los hallazgos alcanzados.

1.1. Reducción de tiempos de compilación mediante arquitectura modular

Uno de los principales objetivos fue reducir los tiempos de compilación al modularizar la aplicación ERP, separando sus componentes en librerías reutilizables y subaplicaciones independientes. Este enfoque coincide con lo planteado por Velepucha and Flores (2021), quienes destacan que la transición hacia arquitecturas modulares o de microservicios, si bien introduce nuevas complejidades, también permite reducir los cuellos de botella asociados a tiempos prolongados de despliegue.

La implementación de subaplicaciones Angular independientes permitió reducir los tiempos de compilación de más de 9 minutos a menos de 3 minutos en los módulos más dinámicos. Esta mejora sustancial se alinea con los beneficios observados por Stricker et al. (2018), quienes señalan que una correcta modularización contribuye a disminuir los tiempos de prueba e implementación, al permitir ciclos de despliegue más ágiles y segmentados.

Con la transición hacia una arquitectura modular, se logró una reducción significativa, como se muestra a continuación:

- Compilación completa del ERP (antes): entre 8:43 y 9:29 minutos.
- Compilación de módulos con librerías: promedio de 5 minutos.
- Compilación de subaplicaciones independientes: entre 1:58 y 2:21 minutos.

1.2. Implementación de pipelines CI/CD eficientes adaptados a la modularidad

El diseño de pipelines CI/CD que compilan solo los módulos afectados fue una pieza clave del modelo propuesto. De manera similar, Pérez López (2023) muestran que la incorporación de herramientas como Jenkins, Docker y Kubernetes permite mejorar la eficiencia del ciclo de vida del software mediante automatización y control de errores, lo cual es coherente con los beneficios alcanzados al integrar GitLab CI, Helm y Kubernetes en esta tesis.

Asimismo, Ferrara et al. (2023) destacan que el uso de plataformas como GitLab para la gestión de configuración y despliegue mejora la coordinación y eficiencia del equipo. Este hallazgo también se evidenció en nuestro proyecto, donde la automatización de compilaciones y despliegues contribuyó a una mayor productividad del equipo de desarrollo, al reducir los tiempos de espera y simplificar el mantenimiento del pipeline.

1.3. Optimización del uso de recursos y reducción de costos operativos

La reducción de tiempos de ejecución del pipeline tuvo un impacto directo en el uso de recursos computacionales. Tal como señalan Abdollahi Vayghan et al. (2018), la optimización de configuraciones en herramientas como Kubernetes es esencial para evitar tiempos de inactividad y sobrecargas de red. En nuestro caso, se logró aprovechar las capacidades de despliegue orquestado mediante Helm, minimizando así los tiempos muertos y permitiendo la entrega continua sin interrupciones.

Además, la simulación de costos en AWS EC2 evidencia que los ahorros por ejecución se amplifican cuando se considera la frecuencia de uso en entornos empresariales. Este análisis coincide con los argumentos de Stricker et al. (2018), quienes resaltan que adoptar modelos más ágiles no solo responde a cuestiones técnicas, sino también estratégicas y organizacionales, permitiendo mejorar la eficiencia y reducir costos operativos.

En el modelo anterior, la compilación redundante de toda la aplicación demandaba recursos considerables de CPU, memoria y almacenamiento temporal en los runners de CI/CD. Con la implementación del nuevo pipeline, los recursos ahora se utilizan de manera más eficiente:

- Solo se compilan los módulos modificados.
- Se reutilizan artefactos compilados desde el caché persistente en el runner.
- Las imágenes Docker se construyen en paralelo, minimizando el tiempo total de construcción.
- Se evitan instalaciones repetitivas de dependencias (node-modules) gracias a enlaces referenciales.

Aunque no se midieron explícitamente las métricas de CPU o memoria, la reducción del tiempo de procesamiento y la menor carga en los runners es un indicativo claro de una disminución en los costos operativos asociados al proceso de CI/CD.

1.4. Reducción de costos de infraestructura en la nube

Aunque el sistema ERP se ejecuta en una infraestructura propia, se realizó una simulación de costos operativos para demostrar el impacto económico que esta optimización tendría en un entorno de nube pública, como AWS.

En este escenario, los procesos de compilación y despliegue continuo pueden beneficiarse del modelo de facturación bajo demanda ofrecido por instancias EC2, donde solo se paga por el tiempo efectivamente utilizado. Este enfoque resulta especialmente adecuado para pipelines CI/CD, los cuales no requieren servidores ejecutándose de manera continua, sino que se activan esporádicamente según los cambios en el código.

De acuerdo con la Tabla 4.7, aunque los montos individuales parecen bajos, su impacto se vuelve significativo al escalar la cantidad de ejecuciones.

Considerando un escenario de 60 ejecuciones mensuales, la reducción del tiempo de ejecución de 10 a 3 minutos disminuye el costo por ejecución de 0.00693 USD a 0.002079 USD, lo que representa un ahorro aproximado del 70 %:

$$\frac{0{,}00693 - 0{,}002079}{0{,}00693} \times 100 \approx 70\,\%$$

En términos mensuales, el costo se reduce de 0.4158 USD a 0.12474 USD, generando un ahorro de 0.29106 USD. Aunque este valor es bajo, sirve como referencia para evidenciar el impacto acumulativo en escenarios de mayor escala.

Al extrapolar este comportamiento a organizaciones con múltiples pipelines y mayor frecuencia de integración, los beneficios económicos se incrementan de manera proporcional al volumen de ejecuciones.

Esta simulación demuestra que, aunque el sistema actualmente no opera en la nube, el enfoque optimizado de CI/CD propuesto está alineado con buenas prácticas modernas de infraestructura elástica y pago por uso. Por tanto, esta solución no solo es aplicable a entornos locales, sino también altamente efectiva y escalable en infraestructuras cloud.

El modelo modular, combinado con un pipeline eficiente y estrategias de construcción paralela, representa una base sólida para adoptar o migrar a arquitecturas más ágiles y económicas en términos de cómputo.

Conclusiones

1. La estrategia transformar la arquitectura monolítica hacia una arquitectura modular y micro-frontend, para identificar cambios en el código fuente y compilar únicamente los módulos afectados resultó ser altamente efectiva. Esto no solo permitió disminuir el uso de recursos computacionales, sino que también simplificó el mantenimiento del pipeline y mejoró la eficiencia del ciclo de desarrollo. Además, la combinación de Angular Libraries y Sub-apps permitió una arquitectura flexible, escalable y orientada al crecimiento futuro de la plataforma.

2. La presente tesis demostró que la implementación de un modelo optimizado de CI/CD adaptado a arquitecturas modulares mejora significativamente los tiempos de compilación, construcción de imagenes docker y despliegue. A través de la modularización de una aplicación ERP real y la aplicación de técnicas como compilación selectiva y despliegue independiente, se logró reducir el tiempo promedio de construcción de 9 minutos a aproximadamente 3 minutos, sin comprometer la estabilidad ni el rendimiento del sistema.

3. La construcción paralela de imágenes Docker y su despliegue por separado mediante Helm y Kubernetes permitió una mayor agilidad en el ciclo de vida del software, facilitando la entrega continua y el aislamiento de funcionalidades críticas. Este enfoque modular no solo mejora la velocidad de entrega en el proceso de ejecución del pipeline optimizado, sino que también contribuye a una mejor trazabilidad de los cambios en producción.

4. Finalmente, el análisis de costos operativos, tanto en entornos locales como en simulaciones en la nube con AWS EC2, evidenció que optimizar los tiempos de ejecución del pipeline tiene un impacto económico positivo cuando se considera la escala de uso en contextos empresariales. Esto valida que la propuesta es no solo técnicamente viable, sino también económicamente sostenible y adaptable a distintos entornos de infraestructura.

Recomendaciones

1. Se recomienda continuar con la evolución hacia una arquitectura completamente desacoplada utilizando micro-frontends y microservicios, lo cual permitiría una mayor independencia entre módulos, mayor velocidad de trabajo para el desarrollador, facilitando su mantenimiento, prueba y despliegue. Esta transición puede ser progresiva, iniciando por los módulos con mayor frecuencia de cambio o criticidad funcional.

2. Es importante complementar el pipeline actual con un sistema automatizado de pruebas (unitarias, integración y end-to-end) para garantizar la calidad del código antes del despliegue. La incorporación de pruebas automáticas ayudará a prevenir errores en producción y facilitará la integración continua en un entorno DevOps completo.

3. Aunque actualmente el entorno de CI/CD se ejecuta en servidores locales, se recomienda implementar pruebas controladas utilizando servicios de cómputo temporales (como máquinas virtuales en la nube). Esto permitiría comparar de forma práctica los beneficios en escalabilidad, tiempos de ejecución y costos reales entre una infraestructura local y una basada en la nube, y serviría como insumo para una posible adopción futura.

4. Por último, se sugiere establecer métricas adicionales de rendimiento y monitoreo continuo sobre el pipeline y los módulos del sistema. Medir el uso de recursos, la frecuencia de cambios por módulo y el impacto en el tiempo de entrega permitirá refinar el modelo actual y adaptarlo a nuevas exigencias técnicas o del negocio con base en datos concretos.

Bibliografía

Abdollahi Vayghan, L., Saied, M. A., Toeroe, M., and Khendek, F. (2018). Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973.

amazon.com (2024). ¿cuál es la diferencia entre imágenes y contenedores de docker? Accedido el 04 de agosto de 2025.

angular.io (2025). ¿qué es angular? Accedido el 04 de agosto de 2025.

AWS (2024). ¿cuál es la diferencia entre la arquitectura monolítica y la de microservicios? Accedido el 04 de agosto de 2025.

cloud.google.com (2025). ¿qué es el cloud computing? Accedido el 04 de agosto de 2025.

Datascientest.com (2022). Gitlab: Saber todo sobre el repositorio git para devops. Accedido el 13 de julio de 2025.

docker.com (2025). ¿qué es contenedores?

earthly.dev (2023). Essential linux terminal commands.

Ferrara, D., Rodriguez, E., Monfroglio, R. L., Robles, M., Díaz Lapérgola, M. A., and Nahuel, L. (2023). Tecnología colaborativa para investigadores: implementación y optimización de un repositorio gitlab para gestión de la configuración y versionado. In *Libro de actas - XXIX Congreso Argentino de Ciencias de la Computación - CACIC 2023*, pages 803–807, Luján, Argentina. Red de Universidades con Carreras en Informática, Institución de origen: Red de Universidades con Carreras en Informática.

GitLab Inc. (2024). Gitlab runner documentation. Accedido el 13 de julio de 2025.

helm.sh (2025). El administrador de paquetes para kubernetes. Accedido el 04 de agosto de 2025.

Hernández Salazar, E. Y. and Beltrán, C. A. (2022). Scrum, un enfoque práctico de metodología ágil para la ingeniería de software. *Tecnología Investigación y Academia*, 8(2).

Hyun, G., Oak, J., Kim, D., and Kim, K. (2024). The impact of an automation system built with jenkins on the efficiency of container-based system deployment. *Sensors*, 24(18):6002.

ibm.com (2024). ¿qué es docker? Accedido el 04 de agosto de 2025.

ibm.com (2025). ¿qué es un registro de contenedores? Accedido el 04 de agosto de 2025.

jqlang.org (2024). jq: sed for json. Accedido el 13 de julio de 2025.

kubernetes.io (2024). ¿qué es kubernetes?

Medium.com (2024). Arquitectura modular: la clave para un desarrollo eficiente de

aplicaciones móviles. Accedido el 04 de agosto de 2025.

medium.com (2024). What is dockerfile.

Oduri, S. (2024). Cloud-native observability and operations: Empowering resilient and scalable applications. *INTERNATIONAL JOURNAL OF ADVANCED RESEARCH IN ENGINEERING AND TECHNOLOGY (IJARET)*, 15(3):323–332.

Pérez López, J. A. (2023). Integración continua y despliegue continuo de una aplicación.

Sampieri, R. H., Collado, C. F., and Lucio, P. B. (2014). Metodología de la investigación 6ta edición. *CF Roberto Hernanadez Sampieri, Metodologia De La Investigacion 6ta edición. MEXICO: McGRAW-HILL*.

Schlichter, J., Klyver, K., and Haug, A. (2021). The moderating effect of erp system complexity on the growth–profitability relationship in young smes. *Journal of Small Business Management*, 59(4):601–626.

Stricker, R., Müssig, D., and Lässig, J. (2018). Microservices for redevelopment of enterprise information systems and business processes optimization. In *ICEIS (2)*, pages 719–726.

tetrate.io (2024). Kubernetes ingress example uses diagram.

Velepucha, V. and Flores, P. (2021). Monoliths to microservices - migration problems and challenges: A sms. In *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pages 135–142.

w3schools.com (2025). Tutorial de bash. Accedido el 04 de agosto de 2025.

xataka.com (2019). De docker a kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo.

Anexos

Cusco, 22 de julio de 2025

Señores:
Ing. Juan Pablo Vizcardo Zuñiga
Gerente General
Ing. Bengie Nick Serrano Quispe
Director de Tecnología
Smart Cities Perú S.A.C

Presente.-

Asunto: Solicitud de emisión de carta de confirmación de actividades realizadas en el marco de proyecto de tesis

De mi consideración:

Yo, **Edwar Yuri Cassa Lipa**, identificado con DNI N.º 70401739, me dirijo a ustedes de manera respetuosa para solicitar la emisión de una carta que confirme las actividades realizadas por mi persona en el marco de mi proyecto de tesis, desarrollado en las instalaciones de la empresa.

Durante mi etapa de colaboración con su organización, desarrollé un modelo de optimización de compilación, integración y despliegue continuo (CI/CD) aplicado al sistema ERP institucional. Asimismo, conté con acceso controlado a los entornos de desarrollo y producción, además del soporte técnico brindado por el equipo de Tecnología de la Información, lo cual fue fundamental para el avance y validación de mi proyecto.

La presente solicitud tiene como finalidad contar con un documento formal que respalde la veracidad de las actividades mencionadas, con fines académicos y para su inclusión en el expediente de mi proyecto de tesis, conforme a los requerimientos establecidos por mi universidad.

Agradezco de antemano su atención y quedo a la espera de su aprobación para proceder conforme al reglamento institucional.

Atentamente,

Edwar Yuri Cassa Lipa
Bachiller en Ingeniería Informática y de Sistemas
DNI: 70401739

Smart Cities Perú S.A.C.
RUC Nº 20601204046
084 – 655042
soporte@smartc.pe
Calle Espinar 1040
Calca - Cusco - Perú

Cusco, 22 de julio de 2025

CARTA N° 015-2025-SCP

Señor:
Edwar Yuri Cassa Lipa
Bachiller en Ingeniería Informática y de Sistemas
Universidad Nacional de San Antonio Abad del Cusco

Presente.-

Asunto: Confirmación de actividades realizadas en el marco de su proyecto de tesis

De nuestra consideración:

Mediante la presente, reciban un cordial saludo. Nos dirigimos a usted para confirmar que lo señalado en su comunicación es correcto. Efectivamente, usted desarrolló en nuestras instalaciones un modelo de optimización de integración y despliegue continuo (CI/CD), aplicado a nuestro sistema ERP, como parte de su proyecto de tesis para optar al título profesional.

Durante este proceso, se le brindó acceso controlado al entorno de desarrollo, así como colaboración técnica por parte de nuestro equipo de Tecnología de la Información, permitiéndole implementar scripts técnicos, ajustar el pipeline de CI/CD y realizar pruebas en entornos locales.

Reiteramos que la información utilizada no incluyó datos confidenciales ni información sensible de clientes o usuarios, y su uso fue estrictamente con fines académicos.

Agradecemos su compromiso y profesionalismo durante su etapa de colaboración con nuestra empresa.

Atentamente,

SMART CITIES PERU S.A.C.
RUC N° 20601204046
Ing. Juan Pablo Vizcardo Zuñiga
GERENTE GENERAL

COLEGIO DE INGENIEROS DEL PERÚ
CONSEJO DEPARTAMENTAL CUSCO
Ing. Bondie Nick Serrano Quispe
INGENIERO INFORMÁTICO Y DE SISTEMAS
CIP 288134