
# Diseño de procesos de Integración Continua mediante la organización de pruebas automatizadas para lograr una retroalimentación oportuna

Andrés Felipe Burbano Castillo

Proyecto de grado entregado para obtener el título de
**Magister en Ingeniería de Software**

Dirigida por
Ph.D. Jaime Alberto Chavarriaga Lozano

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería y Ciencias
Maestría en Ingeniería de Software
Santiago de Cali
28 de enero de 2026

# Ficha Resumen
## Trabajo de Grado Maestría en Ingeniería de Software

**TíTULO:**Diseño de procesos de Integración Continua mediante la organización de pruebas automatizadas para lograr una retroalimentación oportuna

1. Énfasis: Ingeniería de Software
2. Área de trabajo: Aseguramiento de la calidad del software
3. Tipo de proyecto (Aplicado, Innovación, Investigación): Aplicado
4. Estudiante: Andrés Felipe Burbano Castillo
5. Correo electrónico: felipeburbano16@gmail.com
6. Dirección y teléfono: Calle 60A # 120-64, 3127942361
7. Director: Ph.D Jaime Alberto Chavarriaga Lozano
8. Vinculación del director: (Planta/Cátedra/Externo): Cátedra
9. Correo electrónico del director: jaime.chavarriaga@javerianacali.edu.co
10. Co-Director (Si aplica): NA
11. Grupo o empresa que lo avala (Si aplica): Aranda Software
12. Otros grupos o empresas: NA
13. Palabras clave(al menos 5): Automatización de pruebas, pipelines de QA, creación de proceso, retroalimentación continua, pruebas de software, integración continua.
14. ODS que aplica al proyecto (Agenda 2030): NA
15. Fecha de inicio: Mayo 2025
16. Resumen: En la actualidad, muchas organizaciones dependen de pruebas automatizadas y procesos de integración continua para asegurar la calidad de sus productos de software. Sin embargo, en estas empresas suele ocurrir que la ejecución completa de todas las pruebas requiere tiempos prolongados, incluso cuando se hacen cambios pequeños o triviales en el código. Esto genera desmotivación en los desarrolladores, quienes tienden a evitar la ejecución frecuente del pipeline o incluso la creación de nuevas pruebas, por temor a incrementar aún más los tiempos de integración. Esta situación afecta directamente la oportunidad de la retroalimentación, un factor determinante para mantener la calidad y velocidad en ciclos ágiles de desarrollo.

Este proyecto plantea una alternativa un modelo de ejecución segmentada de pruebas automatizadas, en donde no es necesario ejecutar la totalidad de las pruebas en cada evento del flujo de desarrollo, sino solo el subconjunto de pruebas que ofrezca mayor retroalimentación de acuerdo con las actividades que hacen los desarrolladores.

La propuesta parte de un análisis de los diferentes "momentos" del proceso de desarrollo, por ejemplo, la creación de Pull Requests hacia ramas de características (feature) o la integración hacia ramas de liberación (release) y ejecución nocturna completa (nightly build) y define las necesidades de retroalimentación asociadas a cada uno de ellos. A partir de este análisis clasifica las pruebas automatizadas y diseña distintos flujos de integración continua para ejecutar únicamente las pruebas más relevantes para cada etapa, reduciendo así la redundancia y el consumo innecesario de tiempo y recursos.

Como resultado de este proceso se construyó una matriz que relaciona fases del desarrollo, tipos de prueba y tiempos esperados de respuesta, lo cual permitió formalizar un criterio consistente de selección de pruebas. Esta matriz sirvió como base para implementar pipelines YAML parametrizados en Azure DevOps, capaces de seleccionar dinámicamente las pruebas según la rama, el tipo de cambio o el módulo afectado. En los escenarios donde solo se modificaba un módulo específico, la ejecución focalizada mostró una reducción significativa en los tiempos de retroalimentación durante la validación de Pull Requests, sin afectar la calidad de los entregables ni la cobertura esperada de validación.

La evaluación cualitativa, realizada con el jefe del área de DevOps, permitió validar los beneficios percibidos y, a la vez, identificar nuevas oportunidades de mejora. Aunque el filtrado por módulo reduce el tiempo de ejecución de pruebas, se observó que, cuando varios módulos con pipelines independientes se modifican simultáneamente, cada pipeline replica el proceso completo de infraestructura, generando ejecuciones en cascada que incrementan el tiempo total de integración. Este hallazgo destaca la necesidad de evolucionar hacia enfoques basados en orquestadores o infraestructuras compartidas que permitan mantener la segmentación sin incrementar los costos de construcción.

Entre las lecciones aprendidas se destaca la importancia de definir métricas claras de retroalimentación desde el inicio, mantener consistencia en las ramas y disparadores, y minimizar configuraciones manuales para evitar errores humanos. Finalmente, se concluye que la percepción del equipo respecto a la utilidad y fluidez del pipeline es tan relevante como las métricas técnicas al evaluar la efectividad de un proceso de integración continua, especialmente en entornos colaborativos y de alta frecuencia de entrega.

# Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que hicieron posible la culminación de este trabajo.

**Agradecimientos específicos:**

En primer lugar, a mi esposa, quien con su paciencia, compañía y apoyo constante me brindó la motivación necesaria en los momentos en que las fuerzas flaqueaban. A mis padres, por ser el ejemplo de disciplina y perseverancia que me ha guiado a lo largo de mi vida y por recordarme siempre la importancia de alcanzar las metas con dedicación y esfuerzo.

De manera especial agradezco al profesor Jaime Chavarriaga, mi director de tesis, por su orientación, sus observaciones oportunas y el tiempo dedicado a revisar y fortalecer este proyecto. Sus aportes fueron fundamentales no solo en lo académico, sino también en mi formación como profesional.

Extiendo mi gratitud a mis compañeros y colegas, con quienes compartí discusiones y consejos que enriquecieron el desarrollo de esta investigación. Cada conversación y aporte, por pequeño que pareciera, sumó valor al camino recorrido.

Finalmente, agradezco a todas las personas que, de una u otra manera, hicieron parte de este proceso. Concluir esta tesis ha sido un reto personal y profesional que no habría sido posible sin el respaldo, el ánimo y las enseñanzas que recibí en el trayecto.

# Resumen

En la actualidad, muchas organizaciones dependen de pruebas automatizadas y procesos de integración continua para asegurar la calidad de sus productos de software. Sin embargo, en estos entornos, suele ocurrir que la ejecución completa de todas las pruebas requiere tiempos prolongados, incluso cuando se hacen cambios pequeños o triviales en el código. Esto genera desmotivación en los desarrolladores, quienes tienden a evitar la ejecución frecuente del pipeline o incluso la creación de nuevas pruebas, por temor a incrementar aún más los tiempos de integración. Esta situación afecta directamente la oportunidad de la retroalimentación, un factor determinante para mantener la calidad y velocidad en ciclos ágiles de desarrollo.

Este proyecto plantea una alternativa a esta problemática mediante un modelo de ejecución segmentada de pruebas automatizadas, en el cual no es necesario ejecutar la totalidad de las pruebas en cada evento del flujo de desarrollo. La propuesta parte de un análisis de los diferentes "momentos" del proceso, por ejemplo, la creación de Pull Requests hacia ramas de características (feature), la integración hacia ramas de liberación (release) y la construcción nocturna (nightly build), y define las necesidades de retroalimentación asociadas a cada uno de ellos. A partir de este análisis se clasifican las pruebas automatizadas y se diseñan distintos flujos de integración continua que ejecutan únicamente las pruebas más relevantes para cada etapa, reduciendo así la redundancia y el consumo innecesario de tiempo y recursos.

Como resultado de este proceso se construyó una matriz que relaciona fases del desarrollo, tipos de prueba y tiempos esperados de respuesta, lo cual permitió formalizar un criterio consistente de selección de pruebas. Esta matriz sirvió como base para implementar un conjunto de pipelines YAML parametrizado en Azure DevOps, capaces de seleccionar dinámicamente las pruebas según la rama, el tipo de cambio o el módulo afectado.

La propuesta se evalúo en una empresa de desarrollo que construye aplicaciones que se comercializan a nivel mundial y cuyas pruebas pueden tomar varias horas. En esta empresa, en los escenarios donde solo se modificaba un módulo específico, la ejecución focalizada de pruebas mostró una reducción significativa en los tiempos de retroalimentación durante la validación de Pull Requests, sin afectar la calidad de los entregables ni la cobertura esperada de validación.

La evaluación cualitativa, realizada con el jefe del área de DevOps, permitió validar los beneficios percibidos y, a la vez, identificar nuevas oportunidades de mejora. Aunque el filtrado por módulo reduce el tiempo de ejecución de pruebas, se observó que, cuando varios módulos con pipelines independientes se modifican simultáneamente, cada pipeline replica el proceso completo de infraestructura, generando ejecuciones en cascada que incrementan el tiempo total de integración. Este hallazgo destaca la necesidad de evolucionar hacia enfoques basados en orquestadores o infraestructuras compartidas que permitan mantener la segmentación sin incrementar los costos de construcción.

Entre las lecciones aprendidas se destaca la importancia de definir métricas claras de retroalimentación desde el inicio, mantener consistencia en las ramas y disparadores, y minimizar configuraciones manuales para evitar errores humanos. Finalmente, se concluye que la percepción del

equipo respecto a la utilidad y fluidez del pipeline es tan relevante como las métricas técnicas al evaluar la efectividad de un proceso de integración continua, especialmente en entornos colaborativos y de alta frecuencia de entrega.

**Palabras Clave**: Integración continua, Pruebas de software, Diseño de procesos de CI, Retroalimentación continua, Pipelines de CI/CD

# Abstract


## Abstract

Modern software organizations rely heavily on automated testing and continuous integration processes to ensure the quality and stability of their products. However, in real development environments it is common for the complete execution of all automated tests to require significant time, even when the code changes involved are minimal or unrelated to most test cases. This situation often discourages developers from running the pipeline frequently or from creating new tests, as they fear increasing integration times even further. As a result, the opportunity for timely feedback—which is essential in agile development cycles—becomes compromised.

This project proposes an alternative approach through a segmented execution model for automated tests, in which it is not necessary to run the entire test suite at every stage of the development process. The proposed solution begins with an analysis of the different "moments" in the workflow—creating Pull Requests toward feature branches, integrating changes into release branches, and running complete nightly builds—and defines the specific feedback needs associated with each of these moments. Based on this analysis, automated tests were classified and various continuous integration workflows were designed to execute only those tests that are most relevant at each stage, thus reducing redundancy and avoiding unnecessary consumption of time and computational resources.

As part of this work, a matrix was created that relates development phases, test types, and expected feedback times. This matrix served as a foundation for implementing a parametrized YAML configuration in Azure DevOps, capable of selecting tests dynamically according to the branch, the type of change, or the affected module. In scenarios where a single module was modified, the targeted execution of tests showed a significant reduction in feedback time during Pull Request validations, without compromising the quality or coverage expected for the software.

The qualitative evaluation conducted with the DevOps lead confirmed the perceived benefits of the approach while also revealing opportunities for improvement. Although module-based filtering reduced test execution time, it was observed that when multiple modules with independent pipelines were modified simultaneously, each pipeline replicated the complete infrastructure setup process. This behavior produced cascading executions that increased the overall integration time. This finding highlights the need for future iterations of the system to incorporate orchestrators or shared infrastructure strategies capable of maintaining segmentation while avoiding redundant build processes.

Among the lessons learned throughout the project are the importance of defining clear feedback metrics from the beginning, maintaining consistent branch and trigger strategies, and minimizing manual configuration to reduce human error. Ultimately, the study concludes that the team's perception of the pipeline's usefulness and fluidity is just as important as the technical metrics when evaluating the effectiveness of a continuous integration process, especially in collaborative and high-frequency delivery environments.

**Keywords**: Continuous Integration, Software Testing, CI Process Design, Continuous Feedback, CI/CD Pipelines

# Índice general

# Índice de figuras

# Índice de tablas

Capítulo 1

# Introducción

La integración continua (CI) es una práctica de desarrollo de software que busca la detección temprana de errores a través de compilaciones y pruebas ejecutadas de manera frecuente, incluso varias veces al día. Su objetivo es reducir el riesgo de acumular defectos y facilitar que el equipo reciba retroalimentación inmediata sobre el estado del sistema. Sin embargo, en la medida que los proyectos crecen y se acumulan más pruebas automatizadas, los procesos de integración también se hacen más largos (Ron Powell, 2019). En proyectos de gran tamaño, no es extraño que la ejecución completa tarde más de media hora, lo cual desmotiva a los desarrolladores, quienes muchas veces optan por desactivar los pipelines o evitan agregar nuevas pruebas con tal de no extender los tiempos. Esta situación impide alcanzar justamente el beneficio principal de la integración continua: la retroalimentación rápida.

Diversos modelos han tratado de organizar este problema. Uno de los más conocidos es CIViT (Bosch, 2007), que clasifica las pruebas según la cadencia con la que deben ejecutarse (con cada commit, cada hora, diariamente o en pre–despliegue). No obstante, en la práctica, los equipos suelen crear pipelines sin una guía clara, lo que genera redundancia, ejecuciones innecesarias o pruebas críticas que nunca se corren. De allí surge la motivación de este proyecto, que busca ofrecer un método práctico y aplicable en entornos de desarrollo reales.

El objetivo general de este trabajo es diseñar un método para organizar pipelines de integración continua que clasifique las pruebas automatizadas de acuerdo con las necesidades de retroalimentación en cada etapa del proceso de desarrollo. Los objetivos específicos incluyen: i) identificar los momentos del flujo de trabajo donde la retroalimentación resulta más crítica, ii) establecer una clasificación de pruebas según su tipo y valor de retroalimentación, iii) diseñar y parametrizar pipelines en Azure DevOps que ejecuten dinámicamente las pruebas adecuadas en cada fase, y iv) evaluar el impacto del método en términos de tiempo hasta la primera falla, cobertura de pruebas y confiabilidad del proceso.

La investigación se desarrollará bajo un enfoque de Design Science Research, que combina la construcción de un artefacto (el método propuesto y sus componentes técnicos) con la evaluación en un entorno real. Se emplearán herramientas propias del ecosistema .NET y Azure DevOps, junto con convenciones de etiquetado en xUnit para la selección de pruebas. El análisis de resultados se basará en métricas como tiempo de retroalimentación, tasa de fallos detectados y percepción del equipo de desarrollo.

La relevancia de este trabajo radica en que responde a una problemática frecuente en la industria: cómo lograr que la integración continua siga siendo útil y eficiente a pesar del crecimiento de los sistemas y sus pruebas. El método que aquí se propone puede servir como guía para equipos que buscan equilibrar velocidad y calidad, y al mismo tiempo aporta a la literatura académica una

propuesta aplicada que combina modelos conceptuales con herramientas prácticas. Finalmente, este documento está organizado de la siguiente manera: en el Capítulo 1 se presenta la introducción; en el Capítulo 2 se describe la metodología de investigación adoptada; en el Capítulo 3 se desarrolla la propuesta del método y su implementación técnica; en el Capítulo 4 se presentan los resultados obtenidos y el análisis de métricas; y en el Capítulo 5 se exponen las conclusiones, limitaciones, lecciones aprendidas y posibles trabajos futuros.

## 1.1. Definición del problema

En términos de metodología ágil, la realización de actividades de pruebas de software es un reto debido a los frecuentes cambios de código, la velocidad de entrega y las demandas del mercado. Para hacer frente a estos retos, la automatización de las pruebas y la integración continua desempeñan un papel importante a la hora de seguir el ritmo de las necesidades de desarrollo. Ejecutar las pruebas con mayor rapidez y proporcionar una retroalimentación rápida es un elemento clave para el éxito de las actividades de automatización de pruebas. Las estrategias de éxito implican la automatización de pruebas en todos los niveles de software, en todos los niveles de aplicación.

La práctica de integración continua fue popularizada por *Kent Beck* como parte de *Extreme Programming*, a finales de la década de los noventa (los libros se publicaron en 1999 y 2000). En esta práctica, los desarrolladores integran el código fuente con frecuencia y cada integración es verificada por un sistema que construye el código y lo prueba automáticamente (Fowler, 2006). El mismo Kent Beck creó los frameworks de xUnit (como SUnit en 1980 y JUnit en la década de los 2000s) para implementar integración continua y test-driven design (TDD) como parte de los procesos de desarrollo.

La Técnica de Visualización de Integración Continua *CIViT* es una forma útil de visualizar todas las actividades de prueba que realiza un equipo de desarrollo (Bosch, 2007). En esta técnica se crean diagramas que muestran, en cada momento del proceso de desarrollo, que tipo de pruebas se realizan sobre los diferentes tipos de elementos del software (p.ej., componentes, subsistemas o sistemas). Estos diagramas del modelo CIViT, no solo permiten visualizar las diferentes actividades, sino también establecer y ajustar el conjunto de pruebas que se debe ejecutar en cada momento.

Sin embargo, a pesar de su utilidad, el modelo CIViT no contempla el uso de pipelines para ejecutar estas pruebas ni un proceso concreto para lograr una retroalimentación a los desarrolladores que sea apropiada de acuerdo al tipo de actividades que se están realizando. Además, el modelo CIViT exhibe las siguientes limitaciones:

- No hay una identificación clara de qué tipo de pruebas realizar ya que no existe un proceso de integración continua que permita realizar ejecuciones de pruebas automatizadas a través de pipelines en diferentes niveles como pruebas unitarias, de integración y regresión. La falta del proceso de integración continua en pruebas además dificulta realizar los diferentes despliegues en los ambientes de desarrollo, staging y producción, ya que el equipo de desarrollo no cuenta con la retroalimentación apropiada y oportuna sobre el proceso de pruebas, lo que por consiguiente afecta la velocidad del desarrollo y entrega del producto (Downs et al., 2010).

Como se plantea en el modelo CIViT, la integración continua presenta retos de tiempos que se asocian a la gestión de un flujo constante de solicitudes de ejecución de pruebas (Rothermel, 2018).

- Los equipos de desarrollo no tienen claro un proceso de integración continua en pruebas: no existe una definición clara sobre la frecuencia de la ejecución, el tipo de pruebas automatizadas en los diferentes pipelines ni cómo se deben realizar. En consecuencia, la calidad del software pasa a un segundo plano (Bosch, 2022), según la pirámide de Mike Cohn (Cohn, 2009a).

- De acuerdo con un estudio a gran escala sobre la evolución de la cobertura de las pruebas (Hilton et al., 2018), los equipos de desarrollo no cuentan con la cobertura necesaria para asegurar la calidad del software. En particular, la cobertura no contempla todas las casuísticas del negocio, de modo que cada modificación en los componentes puede introducir defectos en cadena. Para aumentar la cobertura existen diferentes técnicas de prueba útiles para asegurar la calidad del software; según un estudio de la Universidad Politécnica de Madrid (Peño, 2015), estas técnicas son útiles de acuerdo al contexto en el que se deseen implementar. Entre ellas se incluyen técnicas estáticas (análisis estático del código, revisión técnica) y técnicas dinámicas (caja blanca: pruebas de ruta básica, pruebas de ciclos o bucles, pruebas de condición y condición múltiple; caja negra: partición de equivalencia, análisis de valores límite, pruebas de tabla de decisión).

En síntesis, los equipos de desarrollo enfrentan procesos de integración continua que consumen demasiado tiempo, carecen de criterios claros para decidir qué pruebas ejecutar en cada etapa y no ofrecen retroalimentación oportuna, lo que afecta tanto la calidad como la productividad del desarrollo de software.

### 1.1.1. Formulación del Problema

El problema que aborda esta investigación es la falta de un método que permita clasificar las pruebas automatizadas y organizar su ejecución dentro de pipelines de integración continua, de manera que los equipos de desarrollo reciban retroalimentación oportuna, consistente y útil sin afectar la productividad.

### 1.1.2. Planteamiento del problema

En el contexto del desarrollo ágil, la ejecución de pruebas de software representa un reto constante debido a los frecuentes cambios de código, la presión por ciclos de entrega más cortos y las demandas del mercado. Para enfrentar estos retos, la automatización de pruebas y la integración continua se han convertido en pilares fundamentales, ya que permiten acelerar la validación del software y entregar retroalimentación oportuna al equipo de desarrollo. Sin embargo, cuando la cantidad de pruebas crece, el tiempo de ejecución de los pipelines de integración también aumenta, llegando en algunos casos a ser tan extensos que los desarrolladores optan por deshabilitarlos, lo

que afecta la retroalimentación inmediata y, en consecuencia, la calidad del producto (Ron Powell, 2019; Fowler, 2006).

Si bien modelos como *CIViT* facilitan la visualización de las pruebas en diferentes momentos del ciclo de vida, estos no consideran mecanismos para garantizar retroalimentación rápida y consistente (Bosch, 2007). Estudios recientes confirman que las limitaciones en cobertura, priorización de pruebas y tiempos de ejecución generan retrasos significativos y afectan directamente la productividad de los equipos (Hilton et al., 2018; Downs et al., 2010). Por ejemplo, en escenarios donde no existe una definición clara de la frecuencia de ejecución ni del tipo de pruebas a ejecutar (unitarias, integración o regresión), los equipos suelen relegar la calidad del software a un segundo plano, priorizando únicamente la entrega funcional (Bosch, 2022; Cohn, 2009a).

La investigación actual ha abordado este problema desde diferentes perspectivas. Machalica et al. proponen la selección predictiva de pruebas para reducir el tiempo de ejecución en pipelines extensos (Machalica et al., 2019), mientras que Mehta et al. presentan técnicas de selección de pruebas basadas en datos a gran escala (Mehta et al., 2021a). Otros enfoques buscan optimizar la ejecución paralela en lotes (Fallahzadeh et al., 2023; Li et al., 2023), o mejorar la priorización mediante metodologías conscientes del sistema de construcción (Elsner et al., 2022). Asimismo, Kolawole y Fakokunde destacan la importancia de la integración continua y despliegue como habilitadores de agilidad en entornos DevOps (Kolawole and Fakokunde, 2024). No obstante, muchos de estos enfoques siguen enfrentando retos relacionados con la cobertura de pruebas y la gestión de la retroalimentación, lo que limita su aplicabilidad en contextos reales de ingeniería de software.

En conclusión, la situación actual refleja una brecha entre la necesidad de retroalimentación rápida y confiable en procesos de integración continua, y las soluciones prácticas que permitan balancear cobertura, tiempo de ejecución y pertinencia de las pruebas. De no abordarse esta brecha, los equipos de desarrollo continuarán enfrentando demoras en la validación, incremento de defectos en producción y, en última instancia, una disminución en la calidad y competitividad de los productos de software.

### 1.1.3. Sistematización del Problema

A partir de este planteamiento, se deriva la siguiente pregunta de investigación:

> ¿Cómo diseñar un proceso de integración continua que permita organizar diferentes conjuntos y tipos de pruebas automatizadas de forma que ofrezcan retroalimentación oportuna a los desarrolladores?

De esta pregunta central surgen los siguientes interrogantes específicos que orientan la investigación:

1. ¿Qué necesidades de retroalimentación deben cubrirse en cada etapa del proceso de desarrollo de software?
2. ¿Cómo clasificar las pruebas automatizadas (unitarias, de integración, UI y de performance) de acuerdo con el valor de retroalimentación que ofrecen?

3. ¿Qué diseño de pipelines de integración continua permite equilibrar rapidez de ejecución, cobertura de pruebas y confiabilidad en los resultados?

### 1.1.4. Síntomas, Causas y Efectos

- **Síntomas:** pipelines lentos, pruebas redundantes, desarrolladores desincentivados a correr tests.
- **Causas:** ausencia de estrategia clara de selección de pruebas, falta de clasificación por relevancia y cadencia, y falta de optimización en la ejecución.
- **Efectos futuros si no se actúa:** incremento de defectos en producción, aumento del costo de mantenimiento, deterioro en la cultura de calidad y entregas más lentas.

## 1.2. Objetivos del proyecto

### 1.2.1. Objetivo General

Proponer un método de diseño de procesos de integración continua que permita organizar conjuntos de pruebas automatizadas y su ejecución en diferentes momentos del ciclo de desarrollo (*feature*, *release* y *nightly*), garantizando retroalimentación oportuna y eficiente a los desarrolladores.

### 1.2.2. Objetivos específicos

1. Identificar y levantar los requisitos del proceso de desarrollo de software, comprendiendo las distintas etapas, actividades y puntos de validación donde se requiere retroalimentación.
2. Analizar las estrategias actuales de gestión de ramas en el repositorio y los flujos de ejecución asociados a los diferentes momentos del ciclo de desarrollo (*feature branch*, *release branch* y *nightly build*).
3. Diseñar y configurar pipelines de integración continua que integren políticas de ramas, segmentación de pruebas y ejecución selectiva según el momento de ejecución, utilizando herramientas nativas de Azure DevOps como `VSTest@3`.
4. Evaluar la efectividad del método propuesto analizando los resultados obtenidos con la percepción cualitativa durante su aplicación sobre un módulo real del proyecto, considerando métricas como parametrización de las pruebas, el nivel de retroalimentación entregada al equipo durante los distintos momentos de integración.

## 1.3. Delimitaciones y alcances

Este proyecto de investigación tuvo como objetivo definir un método para el diseño de procesos de integración continua mediante pipelines que ejecutaron diferentes tipos de pruebas en distintos momentos del proceso de desarrollo. El trabajo se limitó a la clasificación de pruebas automatizadas, la definición de reglas de ejecución basadas en la estrategia de ramas y la implementación de pipelines en Azure DevOps para evaluar la retroalimentación obtenida.

El alcance de este proyecto incluyó:

- La identificación de las necesidades de retroalimentación en cada etapa del ciclo de desarrollo.
- El análisis de estrategias de gestión de ramas y su impacto en la integración continua.
- El diseño de pipelines parametrizados capaces de seleccionar dinámicamente las pruebas a ejecutar.
- La evaluación del método propuesto a partir de métricas de retroalimentación y la percepción de los desarrolladores.

Quedaron fuera del alcance de este trabajo de grado:

- Implementaciones fuera del contexto específico de la investigación. No obstante, los pipelines diseñados podrían aplicarse en otros escenarios sin que esto implicara participación, responsabilidad o soporte de este proyecto.
- El soporte, la continuidad o la evolución de la solución después de la entrega del trabajo de grado.
- El desarrollo de nuevas funcionalidades en el software del cliente o la empresa vinculada al proyecto.
- La evolución posterior, nuevas versiones o ampliaciones del conjunto de pipelines y pruebas implementadas como parte de la investigación.

## 1.4. Justificación del trabajo de grado

La automatización de los procesos de integración continua ha demostrado ser una práctica que contribuye a detectar errores de manera temprana y a reducir el costo de corregir defectos en etapas avanzadas del desarrollo (Hilton et al., 2018; Kolawole and Fakokunde, 2024). Este proyecto plantea una metodología que organiza la ejecución de pruebas en tres momentos estratégicos del ciclo de desarrollo —*feature branch*, *release branch* y *nightly build*—, lo que permite equilibrar la velocidad de retroalimentación y la cobertura de validación.

### Beneficios para la Empresa

Desde la perspectiva empresarial, la investigación aportó un modelo que facilitó implementar procesos de integración continua más eficientes, capaces de reducir los tiempos de validación en *pull requests* y de anticipar defectos antes de llegar a producción. Esto generó beneficios económicos al disminuir los sobrecostos asociados con fallos tardíos y al optimizar los recursos de ejecución de pruebas.

### Beneficios para la Universidad

Para la Universidad, contará con un caso de estudio actualizado que refleja las prácticas modernas de integración continua aplicadas a entornos ágiles y distribuidos. Este trabajo servirá como referencia para futuras investigaciones sobre optimización de pruebas automatizadas, segmentación de pipelines y estrategias DevOps orientadas a la mejora continua.

### Beneficios para el Autor

Finalmente, para el autor representó una oportunidad de profundización profesional. El proyecto representa la oportunidad de consolidar experiencia en la implementación práctica de pipelines segmentados, manejo de políticas de ramas y ejecución selectiva de pruebas automatizadas en Azure DevOps. Asimismo, fortalece sus competencias en automatización de pruebas, diseño de flujos CI/CD y uso de herramientas de control de calidad en la nube, integrando teoría, práctica y análisis empírico en un mismo marco metodológico.

## 1.5. Metodología de la investigación

La metodología de este trabajo se basó en la *Investigación Científica del Diseño* (Design Science Research, DSR), un enfoque ampliamente utilizado en ingeniería de software para la construcción de artefactos que resuelven problemas prácticos de un dominio específico (vom Brocke et al., 2022). Bajo este enfoque, la solución se desarrolló mediante iteraciones en las que se diseñó, construyó y evaluó un método para organizar pipelines de integración continua orientados a ofrecer retroalimentación oportuna.

El proceso metodológico se articuló con los objetivos de la investigación y comprendió las siguientes actividades:

1. **Identificación del problema y motivación:** se analizaron las limitaciones de los pipelines de integración continua existentes y se construyó el marco teórico y el estado del arte correspondiente, con el fin de fundamentar el problema detectado.

2. **Definición de objetivos de diseño:** a partir del análisis anterior, se establecieron metas específicas en cada iteración, alineadas con los objetivos de investigación, como la clasificación de pruebas automatizadas y la organización de los pipelines.

3. **Diseño y desarrollo del artefacto:** se propuso y construyó el método de diseño de pipelines, empleando Azure DevOps y convenciones de etiquetado en xUnit para ejecutar dinámicamente distintos tipos de pruebas.

4. **Demostración:** se implementaron pipelines funcionales en un proyecto .NET, mostrando cómo el método permitía seleccionar y ejecutar subconjuntos de pruebas en diferentes momentos del ciclo de desarrollo.

5. **Evaluación:** La validación del método se realizó aplicándolo sobre un módulo específico del sistema y analizando su comportamiento durante los momentos clave del flujo de integración. Para ello, se midieron aspectos prácticos como los tiempos de ejecución de las pruebas y la pertinencia de los resultados entregados en cada fase. Además del análisis técnico, se llevó a cabo una entrevista con el jefe del área de DevOps, quien evaluó la claridad, utilidad y percepción general de la retroalimentación obtenida tras la aplicación del método. Esta combinación entre resultados observables y valoración cualitativa permitió determinar de manera realista qué tan efectivo era el enfoque propuesto dentro del contexto operativo actual de la empresa.

6. **Comunicación de resultados:** los hallazgos se documentaron en este trabajo, destacando las mejoras alcanzadas, las lecciones aprendidas y las oportunidades de evolución del método propuesto.

Este enfoque permitió no solo construir un artefacto aplicable en un contexto real de desarrollo ágil, sino también validar su aporte mediante métricas objetivas y evidencia cualitativa obtenida de los usuarios involucrados.

## 1.6. Resultados obtenidos

Los resultados obtenidos durante el desarrollo del proyecto permitieron comprobar que el método propuesto para organizar las pruebas automatizadas y distribuir su ejecución en momentos específicos del flujo de integración continua sí genera un impacto positivo en la práctica. Aunque la implementación se llevó a cabo sobre un módulo particular del sistema, esto fue suficiente para observar mejoras tangibles en la forma en que el equipo recibe retroalimentación al momento de integrar cambios. En primer lugar, se logró validar que la clasificación de las pruebas —entre unitarias, de integración y de Extremo a Extremo (E2E)— facilita identificar qué tipo de validaciones son realmente necesarias en cada momento del proceso. Esta clasificación, aplicada durante las ejecuciones de prueba realizadas como parte del proyecto, permitió reducir el tiempo total requerido para obtener una señal clara sobre el estado del módulo modificado. En la práctica, esto significa que los desarrolladores ya no necesitan esperar a que se ejecuten pruebas que no aportan información relevante para el cambio realizado, lo que reduce la fricción habitual que se genera en los ciclos de integración. Adicionalmente, las evaluaciones realizadas evidenciaron que la retroalimentación entregada al equipo se volvió más oportuna y comprensible. Gracias a la definición previa de los

tipos de prueba y su propósito, la información producida durante las ejecuciones fue más fácil de interpretar y permitió detectar fallas o inconsistencias con mayor rapidez. Este punto fue especialmente relevante durante la entrevista realizada con el jefe del área de DevOps, quien resaltó que la propuesta no sólo redujo el tiempo de validación, sino que también aportó claridad sobre lo que está ocurriendo en cada prueba y por qué se ejecuta. Desde el punto de vista técnico, el método también permitió evidenciar que los cambios aplicados en un módulo no siempre requieren una ejecución completa de todas las pruebas disponibles. Esta comprensión, respaldada tanto por los resultados prácticos como por la percepción del evaluador, ayudó a confirmar que una organización más estratégica de las pruebas contribuye a disminuir el desgaste operativo sin sacrificar calidad. Si bien el proyecto no buscó reemplazar la ejecución total de pruebas en fases críticas, sí demostró que es posible optimizar los ciclos diarios de integración mediante decisiones más inteligentes sobre qué ejecutar y cuándo. Por último, el proceso permitió identificar oportunidades de mejora que serán útiles para futuros trabajos. Entre ellas, se encuentra la necesidad de contar con mejores herramientas de trazabilidad entre módulos y pruebas, así como mecanismos más estandarizados para asegurar que la clasificación de pruebas se mantenga consistente con el tiempo. No obstante, estas oportunidades no desvirtúan el valor del método, sino que señalan áreas donde su adopción podría ampliarse y profundizarse en fases posteriores. En conjunto, los resultados indican que el método propuesto es viable, útil y alineado con las necesidades reales del entorno de desarrollo evaluado, aportando mejoras concretas tanto en la eficiencia del proceso de integración como en la comprensión de los resultados de las pruebas automatizadas.

Capítulo 2

# Marco de referencia

## 2.1. Integración Continua

Durante un tiempo considerable, los equipos de desarrollo construían software integrando los módulos al final del proceso, lo que ocasionaba errores detectados demasiado tarde, generando reprocesos, retrasos y frustración en los desarrolladores (Duvall et al., 2007). Este fenómeno, conocido como *integration hell*, fue uno de los detonantes para la práctica de la Integración Continua (CI).

La CI se definió como una práctica en la cual los diferentes componentes de software se integraban y se probaban frecuentemente durante el transcurso del proyecto (Fowler, 2006). El término fue introducido por Booch (Booch, 1990) y más tarde popularizado por Beck y Jeffries como parte de *Extreme Programming* (XP) (Beck and Andres, 2004). Posteriormente, metodologías como *Lean Software Development* (Poppendieck and Poppendieck, 2003) y empresas de tecnología como Microsoft (Cusumano and Selby, 1995) incorporaron esta práctica, lo que impulsó la aparición de múltiples herramientas para soportarla.

### 2.1.1. Herramientas de Integración Continua

Las primeras herramientas de CI eran soluciones instaladas *on-premises*, que se conectaban con sistemas de control de versiones y ejecutaban procesos de integración una o varias veces al día. Estas corridas, conocidas como *nightly builds*, consistían en compilar el código, ejecutar pruebas y generar reportes básicos. Ejemplos tempranos incluyen (ThoughtWorks Open Source, 2001), (Kawaguchi, 2008) y (Jenkins Project, 2011).

Con la adopción de la nube, las herramientas de CI evolucionaron hacia servicios integrados con repositorios Git, ofreciendo mayor flexibilidad y variedad de eventos disparadores. Entre los más usados se encuentran (Travis CI, 2011), (Circle Internet Services, Inc., 2014), (AppVeyor Systems Inc., 2014), (GitLab Inc., 2015) y (GitHub, 2019). Estos servicios facilitaron la adopción de CI en equipos distribuidos y en organizaciones con prácticas de DevOps (Mens et al., 2023).

### 2.1.2. Pipelines de Integración Continua

Con el fin de poder brindar una retroalimentación, los procesos de CI deben realizar tareas como: descargar el código fuente, construir el software y ejecutar pruebas automatizadas. Estas tareas deben ser especificadas por los desarrolladores usando *scripts*, flujos de tareas o *pipelines*.

En la actualidad no existe un único lenguaje o forma de especificación de los pipelines de CI. Basicamente, cada herramientas de CI provee su propio lenguaje y estilo de especificación (Vasilescu et al., 2015).

El script 1 muestra la especificación de un pipeline para Azure DevOps. Allí se puede ver que cada pipeline incluye una serie de *tareas*, y cada tareas tiene una serie de *pasos*. Cada uno de los pasos, a su vez, ejecuta herramientas para realizar las diferentes tareas.

```yaml
#Deshabilita los triggers automáticos para commits directos
trigger:
  branches:
    exclude:
      - '*'
pr:
  branches:
    include:
      - main

#Compila en Windows con dotnet y ejecuta pruebas con VSTest
pool:
  vmImage: 'windows-latest'

steps:

#Instala .NET SDK 9.x
- task: UseDotNet@2
  displayName: 'Install .NET SDK 9.x'
  inputs:
    packageType: sdk
    version: '9.x'
    includePreviewVersions: false

#Instala runtime de .NET 8.0 necesario para ejecutar testhost.exe
- task: UseDotNet@2
  displayName: 'Install .NET Runtime 8.0.x'
  inputs:
    packageType: runtime
    version: '8.0.x'

#Lista los archivos...
- script: dir $(System.DefaultWorkingDirectory) /s /b
  displayName: 'List Files in Directory ...'

#Compila el proyecto
- task: DotNetCoreCLI@2
  displayName: 'dotnet build'
  inputs:
    command: 'build'
    projects: 'v3/Samples.v3.sln'
    configuration: '$(buildConfiguration)'

# Ejecuta las pruebas con VSTest y filtra por el Trait especificado
- task: VSTest@3
  displayName: 'Test execution'
```

```
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: |
      **/TestOrderExamples.dll
      !**/TestAdapter.dll
      !**/obj/**
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Unitaria'
    codeCoverageEnabled: false

# Publica los resultados de las pruebas
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testRunTitle: 'Test Results'
    publishRunAttachments: true
  condition: succeededOrFailed()
  displayName: 'Publish Test Results'
```

**Archivo yml 1:** Pipeline Pull Request feature branch

```
#Deshabilita los triggers automáticos para commits directos
trigger:
  branches:
    exclude:
      - '*'
pr:
  branches:
    include:
      - main

#Compila en Windows con dotnet y ejecuta pruebas con VSTest
pool:
  vmImage: 'windows-latest'

steps:

#Instala .NET SDK 9.x
- task: UseDotNet@2
  displayName: 'Install .NET SDK 9.x'
  inputs:
    packageType: sdk
    version: '9.x'
    includePreviewVersions: false

#Instala runtime de .NET 8.0 necesario para ejecutar testhost.exe
- task: UseDotNet@2
  displayName: 'Install .NET Runtime 8.0.x'
  inputs:
```

```yaml
    packageType: runtime
    version: '8.0.x'

#Lista los archivos...
- script: dir $(System.DefaultWorkingDirectory) /s /b
  displayName: 'List Files in Directory ...'

#Compila el proyecto
- task: DotNetCoreCLI@2
  displayName: 'dotnet build'
  inputs:
    command: 'build'
    projects: 'v3/Samples.v3.sln'
    configuration: '$(buildConfiguration)'

# Ejecuta las pruebas con VSTest y filtra por el Trait especificado
- task: VSTest@3
  displayName: 'Test execution'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: |
      **/TestOrderExamples.dll
      !**/TestAdapter.dll
      !**/obj/**
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Unitaria|Tipo=Integracion'
    codeCoverageEnabled: false

# Publica los resultados de las pruebas
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testRunTitle: 'Test Results'
    publishRunAttachments: true
  condition: succeededOrFailed()
  displayName: 'Publish Test Results'
```

**Archivo yml 2:** Pipeline Pull Request release branch

```yaml
#Deshabilita los triggers automáticos para commits directos
trigger:
  branches:
    exclude:
      - '*'
pr:
  branches:
    include:
      - main

```

```yaml
#Compila en Windows con dotnet y ejecuta pruebas con VSTest
pool:
  vmImage: 'windows-latest'

steps:

#Instala .NET SDK 9.x
- task: UseDotNet@2
  displayName: 'Install .NET SDK 9.x'
  inputs:
    packageType: sdk
    version: '9.x'
    includePreviewVersions: false

#Instala runtime de .NET 8.0 necesario para ejecutar testhost.exe
- task: UseDotNet@2
  displayName: 'Install .NET Runtime 8.0.x'
  inputs:
    packageType: runtime
    version: '8.0.x'

#Lista los archivos...
- script: dir $(System.DefaultWorkingDirectory) /s /b
  displayName: 'List Files in Directory ...'

#Compila el proyecto
- task: DotNetCoreCLI@2
  displayName: 'dotnet build'
  inputs:
    command: 'build'
    projects: 'v3/Samples.v3.sln'
    configuration: '$(buildConfiguration)'

# Ejecuta las pruebas con VSTest y filtra por el Trait especificado
- task: VSTest@3
  displayName: 'Test execution'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: |
      **/TestOrderExamples.dll
      !**/TestAdapter.dll
      !**/obj/**
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Performance'
    codeCoverageEnabled: false

# Publica los resultados de las pruebas
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testRunTitle: 'Test Results'
```

```
    publishRunAttachments: true
  condition: succeededOrFailed()
  displayName: 'Publish Test Results'
```

**Archivo yml 3:** Pipeline midnight build main branch

```yaml
# Nightly build: ejecución completa fuera de horario laboral
schedules:
- cron: "0 0 * * *"   # 00:00 todos los días (UTC)
  displayName: Nightly full run
  branches:
    include:
      - main
  always: true
trigger:
  branches:
    exclude:
      - '*'
pool:
  vmImage: 'windows-latest'
steps:
- task: UseDotNet@2
  displayName: 'Install .NET SDK 9.x'
  inputs:
    packageType: sdk
    version: '9.x'

- task: UseDotNet@2
  displayName: 'Install .NET Runtime 8.0.x'
  inputs:
    packageType: runtime
    version: '8.0.x'

- task: DotNetCoreCLI@2
  displayName: 'dotnet build (nightly)'
  inputs:
    command: 'build'
    projects: 'v3/Samples.v3.sln'
    configuration: 'Release'

# Unitaria
- task: VSTest@3
  displayName: 'Unit tests (Tipo=Unitaria)'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: '**/*.Tests.dll'
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Unitaria'

```

```yaml
# Integración
- task: VSTest@3
  displayName: 'Integration tests (Tipo=Integracion)'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: '**/*.Tests.dll'
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Integracion'

# Regresión
- task: VSTest@3
  displayName: 'Regression tests (Tipo=e2e)'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: '**/*.Tests.dll'
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=e2e'

- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testRunTitle: 'Nightly - Full Suite'
    publishRunAttachments: true
```

### 2.1.3. Pruebas Automatizadas en la Integración Continua

Las pruebas automatizadas son el núcleo de CI, pues permiten validar continuamente la calidad del software. Aunque requieren esfuerzo inicial de desarrollo, aportan beneficios en reducción de costos y mejora de calidad cuando se ejecutan de forma recurrente (Piattini et al., 2013; Mehta et al., 2021b). Estas pruebas pueden implementarse con frameworks como JUnit, NUnit o xUnit, o mediante herramientas de prueba de servicios y UI como JMeter o Postman.
El modelo de la pirámide de pruebas (Cohn, 2009b) establece tres niveles: unitarias, integración/-servicios y UI, siendo las pruebas unitarias la base más amplia. Adicionalmente, el ISTQB clasifica las pruebas en funcionales, no funcionales, de regresión y de aceptación, proporcionando un marco más completo (van Veenendaal, 2018). Estudios recientes destacan que una mayor madurez en la automatización de pruebas se asocia positivamente con la calidad del producto y con ciclos de entrega más cortos (Wang et al., 2022). Asimismo, el *continuous testing* se consolidó como práctica esencial para reducir riesgos en producción y acelerar la entrega de software (Anonymous, 2023).

## 2.2. Antecedentes

En los últimos años, la literatura científica ha abordado diversas estrategias para optimizar los procesos de integración continua (*Continuous Integration, CI*) y ofrecer retroalimentación más oportuna a los desarrolladores. Entre las aproximaciones más relevantes se encuentran la priorización predictiva de pruebas (Machalica et al., 2019), la selección de pruebas a gran escala basada en

datos (Mehta et al., 2021a) y la optimización de pruebas conscientes del sistema de construcción (Elsner et al., 2022). Fallahzadeh et al. (Fallahzadeh et al., 2023) propusieron acelerar la ejecución de pipelines mediante *batching* dinámico, mientras que Schwendner et al. (Schwendner et al., 2025) aplicaron técnicas de aprendizaje por refuerzo para seleccionar pruebas relevantes, reduciendo significativamente los tiempos de ejecución.

A pesar de estos avances y en los últimos años, múltiples investigaciones han abordado la necesidad de visualizar y optimizar los procesos de integración continua como una forma de mejorar la retroalimentación y la calidad del software. Entre los modelos más influyentes se encuentra el modelo CIViT (*Continuous Integration, Verification, and Testing*), propuesto por (Nilsson et al., 2014), que plantea una estructura temporal para organizar las pruebas automatizadas de acuerdo con el momento en que se ejecutan dentro del ciclo de desarrollo.

### 2.2.1. Modelo CIViT

El modelo CIViT (*Continuous Integration, Verification and Testing*) distingue tres grandes categorías de ejecución de pruebas: (1) pruebas que se ejecutan tras cada cambio o *commit*, (2) pruebas programadas con una periodicidad diaria, y (3) pruebas que se ejecutan antes de la liberación o despliegue del software. En términos sencillos, un *commit* es el registro de un cambio en el repositorio de código; una *build nocturna* es una ejecución programada fuera del horario laboral; y "antes de la liberación" se refiere a la verificación previa a poner el software en producción. Cada categoría provee un punto de control que contribuye a mantener la calidad del sistema de manera continua, estableciendo una correlación directa entre la frecuencia de las pruebas y la oportunidad de la retroalimentación (Nilsson et al., 2014).

A partir de este enfoque, el presente trabajo propone una adaptación práctica del CIViT al contexto de *pipelines* YAML en Azure DevOps, organizando las pruebas automatizadas en tres "momentos" específicos: (a) validaciones rápidas ejecutadas al crear un *Pull Request* (PR) hacia una rama de características (*feature branch*); (b) pruebas de integración cuando se integran cambios en una rama de liberación (*release branch*); y (c) una construcción nocturna (*nightly build*) que ejecuta el conjunto completo de pruebas (unitarias, integración y regresión). En Azure DevOps, las validaciones de PR se implementan como "políticas de rama" que exigen la aprobación de una *build* antes de poder fusionar los cambios, mientras que las ejecuciones nocturnas se modelan con *schedules* en los *pipelines* YAML (Microsoft Learn, 2025, 2024).

Esta adaptación moderna del CIViT no solo facilita visualizar *qué* se prueba *y cuándo*, sino también optimizar la ejecución: prioriza la detección temprana de fallos donde el ciclo de retroalimentación es más corto (PRs), traslada pruebas más costosas a ventanas de baja interferencia (nocturnas) y reserva verificaciones más amplias para los hitos de liberación. La evidencia reciente sugiere que enfoques que *seleccionan* o *priorizan* pruebas en función del cambio logran reducir tiempos de ejecución y acelerar la retroalimentación a desarrolladores (Machalica et al., 2019; Mehta et al., 2021a; Elsner et al., 2022; Fallahzadeh et al., 2023; Hilton et al., 2018). En consecuencia, se establece una convergencia entre los fundamentos teóricos del CIViT y las prácticas actuales de DevOps, reforzando la integración entre calidad, automatización y eficiencia operativa.

El modelo proporciona un escenario para debatir dónde mejorar la estrategia de pruebas y acelerar el proceso de producción (Nilsson et al., 2014). El modelo CIViT se centra en cuatro tipos de pruebas:

- Requisitos funcionales (F).
- Requisitos de calidad (Q).
- Funcionalidad heredada (L).
- Casos extremos (E).

La siguiente figura muestra la distribución de estos elementos en el modelo CIViT.

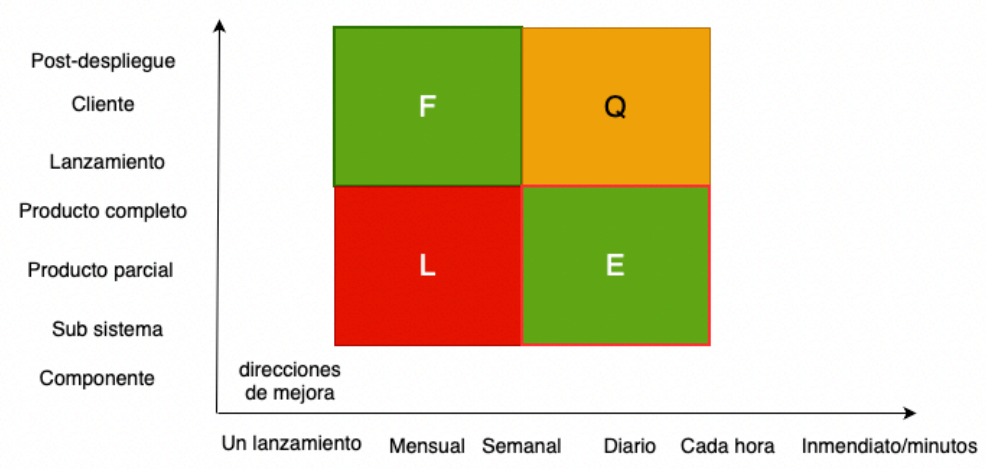



**Figura 2.1:** Disposición de los tipos de actividades de prueba en el modelo CIViT. Elaboración propia

La prueba de funcionalidad se refiere a la prueba del sistema que se está desarrollando actualmente. Por su parte, las pruebas de funcionalidad heredada se refieren a las funciones ya existentes y comprueban si siguen funcionando conforme a los requisitos incluso después de añadir nuevas funcionalidades. Además, las pruebas de atributos de calidad, como el rendimiento, la seguridad, la fiabilidad y los aspectos de seguridad, garantizan que el sistema que se está probando se ajusta a las especificaciones de calidad definidas. Por último, las pruebas de casos extremos garantizan que no se produzcan circunstancias inusuales derivadas de defectos que no se detectaron en la fase inicial y que no se han detectado hasta pasado un tiempo (Nilsson et al., 2014). El color verde dentro del recuadro representa que los requisitos están totalmente cubiertos, mientras que el rojo muestra que no hay cobertura de los requisitos. El color naranja representa que los requisitos están parcialmente cubiertos. Además, la línea del borde del recuadro explica el nivel de automatización en cada fase. El color verde muestra que el nivel está totalmente automatizado, mientras que el rojo ilustra que el nivel se prueba manualmente. El naranja retrata que el nivel está parcialmente automatizado.

## 2.3. Estado del Arte

El estado del arte es la sección en la que se analiza el conocimiento acumulado alrededor de un problema de investigación y se identifican las soluciones existentes, sus ventajas y limitaciones. En este caso, el tema central es el diseño de procesos de Integración Continua (CI, por sus siglas en inglés *Continuous Integration*) que organicen conjuntos de pruebas automatizadas y brinden retroalimentación oportuna a los desarrolladores.
Para un lector no técnico, vale aclarar que **Integración Continua** es una práctica en el desarrollo de software que consiste en integrar cambios de código de manera frecuente en un repositorio compartido. Cada integración dispara automáticamente procesos de compilación y ejecución de pruebas. La meta es detectar errores de forma temprana, antes de que se acumulen y se conviertan en fallos más costosos.
La literatura sobre CI se puede agrupar en tres grandes líneas: (i) **selección y priorización de pruebas** según el cambio realizado, (ii) **estrategias de ejecución** para reducir el tiempo de ciclo, y (iii) **visualización y planificación** de actividades de prueba en distintos momentos del ciclo de vida. A continuación se presentan los avances más relevantes en cada línea, destacando sus aportes y vacíos.

### 2.3.1. Selección y priorización de pruebas

La *selección de pruebas* consiste en decidir qué subconjunto de pruebas ejecutar cuando se modifica el código. La *priorización* determina en qué orden deben ejecutarse para maximizar la probabilidad de detectar fallos tempranamente. Ambas técnicas buscan reducir el tiempo de ejecución sin perder efectividad.
Un referente industrial temprano es el enfoque de *Predictive Test Selection* descrito por Machalica et al. (2019) en Meta (Facebook). En este trabajo se entrenan modelos de aprendizaje automático (*machine learning*) usando el historial de fallos, cambios de código y dependencias. El modelo predice cuáles pruebas son más relevantes para un cambio y selecciona únicamente esas. Así, se logra ejecutar solo una fracción de la suite de pruebas, pero con alta probabilidad de encontrar los defectos. Esto redujo drásticamente los tiempos de CI en proyectos de gran escala sin sacrificar la calidad del producto.
En Microsoft, Mehta et al. (2021a) presentan una solución *data-driven* (es decir, basada en datos históricos) para seleccionar pruebas. Este enfoque integra información de cobertura de código (qué partes del sistema ejercitan las pruebas), historial de fallos y grafo de dependencias del repositorio. El sistema decide de manera automática qué pruebas ejecutar en cada integración, balanceando precisión (ejecutar las pruebas correctas) y recall (no omitir pruebas críticas).
Por su parte, Elsner et al. (2022) plantean un enfoque consciente del sistema de construcción (*build-system aware regression test selection*). En proyectos monorepo o multi-lenguaje, donde el proceso de compilación es complejo, este trabajo considera la forma en que los módulos se construyen y enlazan, para determinar qué pruebas tienen mayor probabilidad de verse afectadas. Esto mejora la precisión de la selección en entornos con múltiples lenguajes y dependencias cruzadas.

**Síntesis.** La evidencia demuestra que la selección y priorización guiadas por datos reducen significativamente los tiempos de retroalimentación en CI. Sin embargo, estas propuestas se enfocan en la decisión de *qué pruebas correr en un momento puntual*, pero no ofrecen un *método de diseño integral* que considere diferentes momentos (p.ej., pruebas rápidas por commit, pruebas de regresión al final del día, pruebas completas antes de liberar).

### 2.3.2. Estrategias de ejecución y reducción del tiempo de ciclo

Además de seleccionar menos pruebas, se han explorado técnicas para *ejecutarlas mejor*. Esto incluye paralelización, orquestación en lotes y optimización de recursos.
Fallahzadeh et al. (2023) introducen el concepto de *Parallel Batch Testing*, donde los casos de prueba se agrupan en lotes que se ejecutan en paralelo en múltiples agentes. Esta estrategia reduce la duración total de la ejecución y balancea la carga de trabajo, especialmente útil cuando hay un gran número de pruebas y recursos limitados.
De manera similar, Li et al. (2023) modelan la ejecución de pruebas en paralelo mediante algoritmos de asignación por lotes, buscando minimizar la latencia global de CI. Para un lector no técnico: se trata de organizar "colas de pruebas" en varios servidores a la vez, para que la respuesta llegue antes.
Estudios cuantitativos en proyectos de código abierto muestran además que una mayor *madurez en la automatización de pruebas* se correlaciona con mayor calidad del producto y ciclos de entrega más cortos. Wang et al. (2022) definen "madurez" como el grado en que los equipos automatizan la creación, ejecución y monitoreo de pruebas, con métricas claras de desempeño. En otras palabras, no basta con tener pruebas; es clave que estén integradas sistemáticamente en el ciclo de desarrollo.

**Síntesis.** Estas propuestas aportan a reducir la duración del ciclo de CI mediante paralelización y optimización de recursos. No obstante, dejan abierta la pregunta de cómo integrar estas técnicas en un método que también considere la necesidad de retroalimentación diferenciada por contexto (p.ej., inmediatez en commits vs. exhaustividad en pre-release).

### 2.3.3. Visualización y planificación de pruebas

Otra línea de investigación se centra en la comprensión y planificación de actividades de prueba mediante modelos de visualización. Un referente es el modelo CIViT (*Continuous Integration and Testing Visualization*) propuesto por Nilsson et al. (2014), que representa gráficamente qué tipos de pruebas se ejecutan en distintos momentos: por commit, por hora, diariamente o antes del despliegue.
El modelo CIViT facilita a los equipos identificar redundancias o huecos en sus estrategias de prueba. Sin embargo, es principalmente una herramienta de visualización conceptual: no prescribe cómo deben diseñarse los pipelines ni cómo medir la efectividad de la retroalimentación. Trabajos posteriores, como Bosch et al. (2016), han destacado la utilidad de estas visualizaciones para alinear prácticas de prueba con arquitecturas ágiles, pero sin ofrecer un marco de diseño operativo.

**Síntesis.** La visualización permite comprender mejor el proceso de pruebas, pero falta articular estas representaciones con mecanismos concretos de selección dinámica, métricas de oportunidad y políticas de pipeline que respondan a las necesidades de los equipos de desarrollo.

### 2.3.4. Comparación de trabajos relevantes

A partir de lo anterior se definieron criterios de comparación: (C1) soporte para selección sensible al cambio, (C2) uso de técnicas de aprendizaje automático o *data-driven*, (C3) conciencia del sistema de construcción y dependencias, (C4) estrategias de ejecución paralela, (C5) medición de oportunidad/efectividad del feedback, (C6) validación en contextos industriales.

**Tabla 2.1:** Comparación de trabajos relevantes en Integración Continua y pruebas automatizadas

| **Referencia** | **Trabajo / Autor** | C1 | C2 | C3 | C4 | C5 | C6 |
|---|---|---|---|---|---|---|---|
| Machalica et al. (2019) | Predictive Test Selection (Meta) | ✓ | ✓ | ○ | ○ | ✓ | ✓ |
| Mehta et al. (2021a) | Data-driven Test Selection at Scale (Microsoft) | ✓ | ✓ | ✓ | ○ | ✓ | ✓ |
| Elsner et al. (2022) | Multi-language Regression Test Selection (TUM & Siemens) | ✓ | ○ | ✓ | ○ | ○ | ○ |
| Fallahzadeh et al. (2023) | Parallel Batch Testing (Concordia Univ.) | ○ | ○ | ○ | ✓ | ✓ | ✓ |
| Wang et al. (2022) | Test Automation Maturity Study (Univ. of Oulu) | ○ | ○ | ○ | ○ | ✓ | ✓ |

*Leyenda*: ✓ = cubierto; ○ = limitado o indirecto.

### 2.3.5. Vacíos identificados

El análisis del estado del arte permite identificar tres vacíos principales:

**V1. Falta de un método integral de diseño.** Los trabajos revisados resuelven partes del problema (selección, paralelización, visualización), pero no integran estas dimensiones en un proceso de diseño completo de pipelines.

**V2. Desconexión con la gestión de ramas.** La mayoría de las propuestas no consideran el contexto de ramas de desarrollo (feature, release, hotfix, main), donde las necesidades de retroalimentación difieren en urgencia y exhaustividad.

**V3. Validación de la configuración.** Pocos estudios abordan cómo verificar que los pipelines realmente se configuren de acuerdo con las políticas y cadencias definidas, lo que deja espacio para inconsistencias.

### 2.3.6. Conexión con esta tesis

Este trabajo aborda los vacíos V1–V3 mediante la propuesta de un método de diseño de procesos de Integración Continua que organiza pruebas automatizadas en conjuntos vinculados a diferentes momentos del ciclo, define políticas explícitas de selección de pruebas según el momento y oportunidad de retroalimentación oportuna. A diferencia de estudios anteriores, se plantea una guía práctica reproducible que conecta las necesidades de los equipos de desarrollo con la configuración de los pipelines en plataformas como Azure DevOps.

## 2.4. Resumen del capítulo

El marco de referencia presenta una visión clara del contexto técnico y científico en el que se desarrolla este trabajo, centrado en la mejora de los procesos de *Integración Continua (CI)* y en la organización eficiente de las pruebas automatizadas. Se describe cómo las primeras aproximaciones a la CI surgieron como respuesta al llamado *integration hell*, donde los equipos de desarrollo integraban código al final del proceso, generando errores y retrasos significativos. A partir de autores como Fowler, Beck y Booch, se establecieron las bases de la CI moderna, que hoy es parte fundamental de las prácticas de *DevOps* en empresas de software de todo el mundo.

El capítulo revisa la evolución de las herramientas y servicios de CI —desde soluciones locales como *CruiseControl* o *Hudson*, hasta plataformas en la nube como *GitHub Actions* o *Azure DevOps*— y cómo estas han permitido automatizar la construcción, ejecución de pruebas y despliegues. Se destaca el papel de las pruebas automatizadas como núcleo del proceso de integración, explicando su clasificación (unitarias, de integración y de interfaz) y su relevancia para garantizar calidad y detectar fallos oportunamente.

Asimismo, se analizan investigaciones recientes que abordan la selección inteligente de pruebas, la reducción de tiempos de ejecución y la visualización del ciclo de pruebas. Modelos como *CIViT* aportan una base conceptual útil, aunque se observa una brecha en la integración de estos enfoques con la gestión de ramas y la medición de la latencia de retroalimentación. De este análisis emergen tres vacíos principales: la falta de un método integral de diseño de pipelines, la desconexión entre la CI y la gestión de ramas de desarrollo, y la ausencia de mecanismos sistemáticos para validar la configuración de pipelines y políticas de ejecución.

En síntesis, este capítulo consolida los fundamentos teóricos y empíricos que sustentan la investigación, mostrando que, a pesar de los avances, aún existe una necesidad clara de metodologías que unifiquen automatización, retroalimentación y control de calidad dentro de un proceso de integración continua más ágil, confiable y adaptable a los entornos modernos de desarrollo de software.

Capítulo 3

# Desarrollo del Proyecto

## 3.1. Diseño de la Propuesta

El diseño de la propuesta se orientó a solucionar una problemática común en los procesos de integración continua: la ejecución completa de todas las pruebas automatizadas ante cualquier modificación del código, sin importar su alcance o impacto real en el sistema. Esta situación ocasionaba una sobrecarga en los pipelines de CI/CD, ya que incluso un cambio mínimo —por ejemplo, un comentario o ajuste menor en un módulo— desencadenaba la ejecución de pruebas en todos los componentes del sistema, generando tiempos de espera prolongados y una retroalimentación tardía para los desarrolladores.

En el contexto de este trabajo, la solución se centró en el diseño de pipelines independientes por módulo dentro del ecosistema de Azure DevOps, utilizando descripciones declarativas en archivos YAML. Cada pipeline fue configurado para ejecutar únicamente el conjunto de pruebas asociado a su módulo específico, garantizando una validación focalizada del código. Esta arquitectura modular permitió detectar fallos más rápidamente y reducir el tiempo promedio de retroalimentación, manteniendo la consistencia del proceso de integración.

Para asegurar la trazabilidad de las ejecuciones, cada pipeline incluía tareas estructuradas en etapas de compilación, prueba y publicación de resultados, empleando las tareas nativas de Azure DevOps como `DotNetCoreCLI@2` para la compilación y `VSTest@3` para la ejecución de pruebas automatizadas. De este modo, se mantuvo una homogeneidad técnica entre los diferentes pipelines y se estandarizó la forma de registrar los resultados y métricas de prueba.

Una observación relevante surgida durante la fase de validación fue que, al ejecutar un solo pipeline con la configuración de filtros adecuada —es decir, parametrizado para detectar y procesar únicamente los cambios de su módulo—, se evidenció una ganancia sustancial en eficiencia. En dichos casos, el pipeline ejecutaba una única vez la infraestructura y las pruebas correspondientes, evitando duplicaciones innecesarias y optimizando los tiempos de ejecución sin afectar la cobertura de calidad. Esta evidencia reforzó la premisa central del proyecto: la automatización inteligente de pruebas debe priorizar la pertinencia y oportunidad de la retroalimentación, no la cantidad de ejecuciones.

En consecuencia, el diseño general de la propuesta se basó en una estructura modular, declarativa y fácilmente escalable. Cada pipeline actúa como una unidad autónoma de validación, que se activa bajo condiciones específicas (por ejemplo, cambios detectados en su módulo correspondiente), manteniendo al mismo tiempo la posibilidad de integrarse en flujos más amplios de validación y despliegue. Este enfoque equilibró simplicidad, eficiencia y control, sentando las bases para posteriores mejoras orientadas a la automatización dinámica y a la orquestación de pipelines.

### 3.1.1. Descripción General

El desarrollo de esta propuesta surgió a partir de una problemática observada durante la gestión de procesos de integración continua en entornos de desarrollo que trabajan con soluciones modulares. A medida que el sistema crecía, se hizo evidente que los pipelines configurados de manera individual para cada módulo generaban redundancia, ya que múltiples flujos realizaban las mismas tareas de compilación y preparación de entorno cuando existían cambios que afectaban más de un componente. Este comportamiento no solo incrementaba los tiempos de ejecución, sino que también reducía la eficiencia global del proceso de integración continua y afectaba la retroalimentación hacia los desarrolladores.

Uno de los principales problemas identificados fue que, ante cualquier modificación en el código —sin importar su tamaño o relevancia—, el pipeline ejecutaba la totalidad de las pruebas automatizadas. Esto incluía tanto las pruebas del módulo afectado como las de otros módulos que no guardaban relación alguna con el cambio. En la práctica, incluso una modificación menor, como agregar un comentario en el código o realizar un ajuste mínimo en una clase, desencadenaba la ejecución completa del conjunto de pruebas. Este comportamiento provocaba que los desarrolladores tuvieran que esperar largos periodos, en algunos casos de hasta treinta minutos, para recibir una retroalimentación sobre el resultado de la ejecución, lo que generaba demoras, pérdida de tiempo y una sensación de ineficiencia en el flujo de trabajo.

En la práctica, cada pipeline ejecutaba de forma secuencial las mismas etapas de instalación, compilación y pruebas, lo que implicaba un uso innecesario de recursos cuando las modificaciones realizadas eran menores o afectaban únicamente una parte específica del código. Si bien este modelo funcionaba correctamente para proyectos pequeños, en un contexto de múltiples módulos con interdependencias, el tiempo invertido en procesos repetidos comenzó a ser significativo. Este escenario motivó la necesidad de buscar un enfoque que permitiera optimizar la ejecución de pruebas, garantizando que solo se validaran los módulos realmente afectados por los cambios.

Además, otro de los retos identificados fue la oportunidad de mejora en la retroalimentación que recibían los desarrolladores. En la configuración inicial, los tiempos de espera entre la ejecución de los pipelines y la publicación de resultados eran prolongados, lo que dificultaba detectar y corregir errores de manera ágil. Esto afectaba directamente la dinámica de trabajo en equipo y el flujo de entrega continua, dos pilares esenciales en las prácticas modernas de *DevOps*.

El objetivo principal de la propuesta, entonces, fue diseñar un proceso que permitiera optimizar la ejecución de los pipelines reduciendo la redundancia en los procesos repetitivos, pero sin perder la trazabilidad y la independencia funcional entre módulos. La solución debía además mantener la capacidad de entregar resultados de pruebas automatizadas de manera oportuna, precisa y contextualizada, para fortalecer el ciclo de retroalimentación entre los equipos de desarrollo y asegurar la calidad del software en cada entrega.

En cuanto al impacto esperado, la implementación de esta propuesta permitiría una reducción notable en los tiempos de ejecución de las pruebas automatizadas, especialmente en escenarios donde los cambios se limitan a uno o pocos módulos. También facilitaría la gestión y el mantenimiento de los pipelines, minimizando los errores derivados de configuraciones manuales y mejorando la

productividad de los equipos técnicos. En última instancia, la propuesta busca consolidar una práctica más madura de integración continua, en la cual la automatización no solo agilice el flujo de trabajo, sino que también mejore la calidad y la estabilidad del producto de software.

La siguiente figura muestra los momentos vs tipos de prueba a ejecutar.

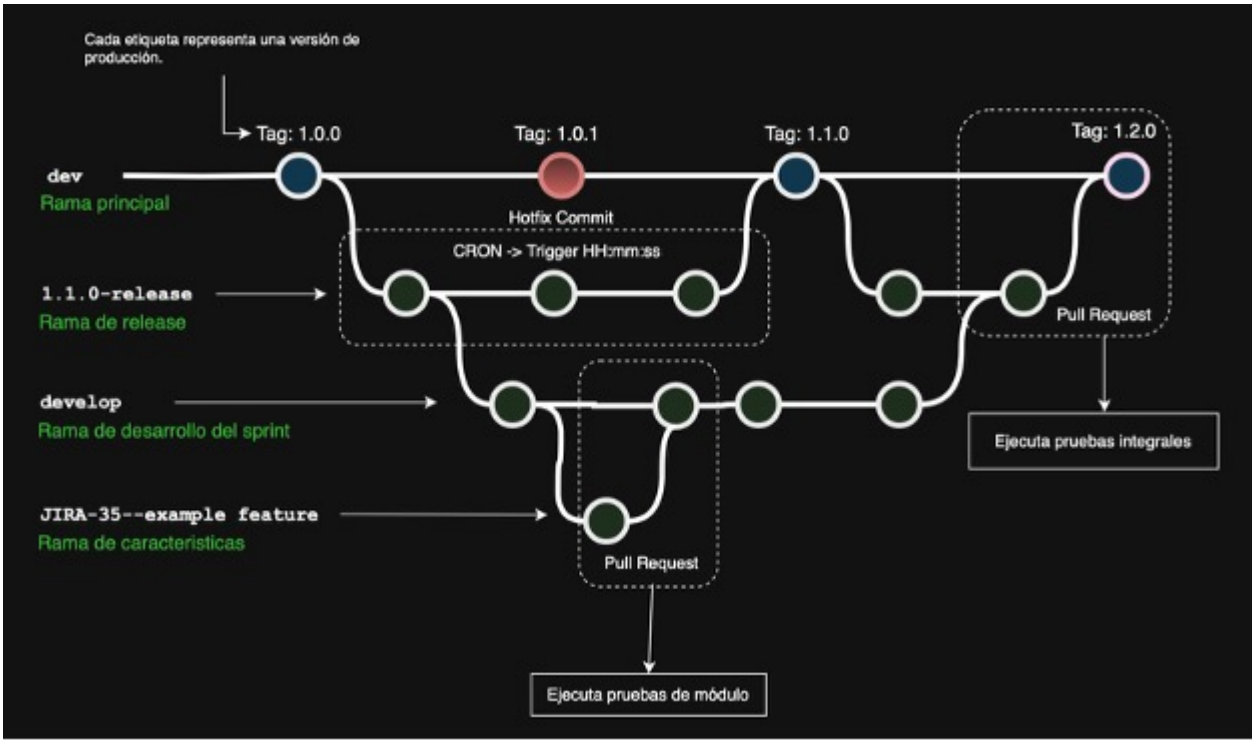



**Figura 3.1:** Momentos de ejecución vs tipos de pruebas a ejecutar. Elaboración propia

### 3.1.2. Usuarios y Beneficiarios

La propuesta desarrollada en este proyecto está dirigida principalmente a optimizar el trabajo de los equipos de desarrollo y de aseguramiento de la calidad en entornos de integración continua, particularmente en contextos donde las soluciones están conformadas por múltiples módulos o componentes de software interdependientes. El sistema propuesto no solo busca mejorar la eficiencia técnica, sino también ofrecer beneficios tangibles a los diferentes roles involucrados en el ciclo de desarrollo de software.

#### Equipos de desarrollo

Los desarrolladores son los principales beneficiarios de la solución. Con la automatización propuesta, reciben retroalimentación más rápida y precisa sobre los cambios realizados en el código. En la configuración tradicional, cualquier modificación —por mínima que fuera— desencadenaba la

ejecución completa de las pruebas automatizadas, lo que implicaba largos tiempos de espera para conocer los resultados. Con la nueva estructura de ejecución segmentada por módulos y la posibilidad de filtrar dinámicamente las pruebas relevantes, los desarrolladores pueden obtener resultados en menor tiempo y enfocarse en resolver problemas específicos del módulo afectado. Esto mejora la productividad individual, reduce la frustración generada por los tiempos de espera innecesarios y promueve una cultura de entrega continua más fluida y ágil.

#### Ingenieros QA y DevOps

Los ingenieros de calidad (QA) y los especialistas en DevOps también se benefician de forma directa. Anteriormente, la configuración manual de múltiples pipelines y las políticas asociadas a cada uno implicaban tareas repetitivas y propensas a error. Cualquier cambio en la estructura del proyecto requería actualizaciones en decenas de archivos YAML y verificaciones manuales para asegurar consistencia entre módulos. La propuesta plantea un enfoque más centralizado y automatizado, reduciendo significativamente la carga operativa y los posibles fallos de configuración. Además, el sistema proporciona una mejor trazabilidad de las ejecuciones de prueba, permitiendo a los equipos de QA enfocar sus esfuerzos en la calidad funcional y no en la gestión del entorno de integración.

#### Organización

A nivel organizacional, los beneficios se reflejan en la optimización de recursos y la reducción de costos indirectos asociados a la ejecución redundante de pipelines. En escenarios donde cada pipeline construía y probaba el sistema completo, los tiempos de procesamiento y el uso de infraestructura se incrementaban de manera considerable. Con la propuesta, se evita la ejecución duplicada de flujos innecesarios, permitiendo un uso más racional de los recursos disponibles en la nube. Esto no solo mejora la eficiencia técnica del entorno de desarrollo, sino que también incrementa la capacidad de la organización para entregar versiones estables con mayor frecuencia y confiabilidad.
En conjunto, esta solución aporta beneficios tanto a nivel técnico como operativo: los equipos de desarrollo obtienen retroalimentación oportuna, los ingenieros QA y DevOps reducen tareas manuales repetitivas y la organización logra una utilización más eficiente de los recursos computacionales. De esta manera, la propuesta contribuye al fortalecimiento de una cultura DevOps orientada a la automatización, la calidad continua y la mejora sostenida de los procesos de integración y entrega de software.

### 3.1.3. Propósito e Impacto Esperado

El propósito central de la propuesta es diseñar un método automatizado que permita orquestar la ejecución de pruebas en Azure DevOps de manera eficiente, reduciendo tiempos, costos y errores humanos. Con ello, se busca contribuir al fortalecimiento de las prácticas DevOps y a la calidad continua del software entregado.

## 3.2. Requisitos del sistema

El presente apartado describe los requisitos funcionales y no funcionales que guiaron el diseño e implementación de la propuesta. Estos requisitos se definieron con base en los objetivos planteados y en la problemática identificada en los procesos de integración continua de la organización, donde la ejecución de pruebas no diferenciaba el tipo ni la relevancia del cambio, generando tiempos de espera prolongados y redundancia en la validación.

### 3.2.1. Requisitos Funcionales

Los requisitos funcionales (RF) definen las capacidades específicas que el sistema debía ofrecer para garantizar la correcta ejecución y segmentación de las pruebas automatizadas en diferentes momentos del ciclo de desarrollo. En la Tabla 3.1 se resumen los principales.

**Tabla 3.1:** Requisitos Funcionales del Sistema

| ID | Descripción |
|---|---|
| RF1 | El sistema debe permitir definir y ejecutar pipelines independientes por módulo dentro del repositorio de código. |
| RF2 | Cada pipeline debe compilar el proyecto utilizando tareas declarativas en YAML con la instrucción `DotNetCoreCLI@2`. |
| RF3 | El sistema debe ejecutar pruebas automatizadas con `VSTest@3`, filtrando por tipo de prueba a través de criterios como `testFilterCriteria`. |
| RF4 | El sistema debe permitir asociar los pipelines a eventos específicos, como la creación de un *Pull Request* hacia una rama de características (*feature branch*). |
| RF5 | El sistema debe ejecutar pruebas de integración cuando se cree un *Pull Request* hacia una rama de liberación (*release branch*). |
| RF6 | El sistema debe ejecutar la totalidad de las pruebas (unitarias, integración y regresión) mediante una construcción programada (*nightly build*). |
| RF7 | El sistema debe registrar y publicar los resultados de las pruebas ejecutadas en cada pipeline mediante la tarea `PublishTestResults@2`. |
| RF8 | El sistema debe permitir identificar los cambios realizados en los archivos del repositorio para ejecutar únicamente las pruebas relacionadas con los módulos afectados. |

### 3.2.2. Requisitos No Funcionales

Los requisitos no funcionales (RNF) establecen las condiciones de calidad, rendimiento y mantenibilidad que el sistema debía cumplir para garantizar su eficacia y adopción por parte del equipo de desarrollo. En la Tabla 3.2 se describen los principales.

**Tabla 3.2:** Requisitos No Funcionales del Sistema

| ID | Descripción |
|---|---|
| RNF1 | La configuración de los pipelines debe realizarse mediante archivos YAML, garantizando su versionamiento y trazabilidad dentro del repositorio. |
| RNF2 | El tiempo promedio de retroalimentación para los desarrolladores no debe superar los 10 minutos en validaciones rápidas (PR hacia *feature branch*). |
| RNF3 | Los pipelines deben ser reproducibles en entornos distintos sin dependencia de configuraciones manuales. |
| RNF4 | El sistema debe garantizar la consistencia en los resultados de prueba, independientemente del entorno de ejecución. |
| RNF5 | La solución debe ser escalable, permitiendo incorporar nuevos módulos o tipos de pruebas sin modificar la estructura general de los pipelines. |
| RNF6 | Las ejecuciones programadas (*nightly builds*) deben realizarse fuera del horario laboral para no afectar la productividad del equipo. |
| RNF7 | La solución debe minimizar el uso redundante de recursos de ejecución, priorizando la eficiencia en tiempos de ejecución y en costos de infraestructura. |
| RNF8 | El diseño de los pipelines debe ser entendible y documentado, facilitando su mantenimiento y la transferencia de conocimiento al equipo DevOps. |

En conjunto, estos requisitos garantizan que la propuesta cumpla su propósito principal: ofrecer una estructura de integración continua más ágil, adaptable y capaz de proveer retroalimentación oportuna a los desarrolladores, sin sacrificar la cobertura ni la confiabilidad de las pruebas.

## 3.3. Diseño de la Solución

La solución propuesta se diseñó con el propósito de optimizar los procesos de integración continua en proyectos de software desarrollados bajo entornos colaborativos. El enfoque central consistió en disminuir la redundancia en la ejecución de pruebas automatizadas y mejorar la retroalimentación a los desarrolladores, manteniendo la trazabilidad y la consistencia de los resultados.
La arquitectura se construyó sobre la plataforma Azure DevOps, utilizando *pipelines* definidos mediante archivos YAML. Estos archivos permiten describir de forma declarativa las etapas de construcción, ejecución de pruebas y publicación de resultados, facilitando el versionamiento y la replicación del flujo en diferentes entornos.
El diseño general de la solución se basa en tres componentes principales: (1) los *pipelines* modulares, que representan la unidad funcional de ejecución; (2) las políticas de rama, encargadas de activar la validación automática de código ante *Pull Requests*; y (3) la configuración de ejecuciones programadas, destinada a ejecutar validaciones integrales fuera del horario laboral.

### 3.3.1. Arquitectura Conceptual

La arquitectura conceptual se representa como un conjunto de *pipelines* distribuidos por módulo, donde cada uno posee la lógica necesaria para construir, ejecutar y reportar los resultados de su propio conjunto de pruebas. Esta segmentación modular permite que la ejecución de pruebas se limite únicamente a los componentes afectados por un cambio, evitando la ejecución completa del conjunto de pruebas del proyecto.
Cada *pipeline* se compone de tres etapas principales:

- **Etapa de compilación:** se utiliza la tarea `DotNetCoreCLI@2` para construir el proyecto o la solución asociada al módulo. Esta tarea garantiza la correcta generación de los artefactos requeridos para la ejecución de pruebas.
- **Etapa de pruebas automatizadas:** implementada mediante la tarea `VSTest@3`, la cual ejecuta pruebas unitarias o de integración según el filtro definido en el parámetro `testFilterCriteria`. Este filtro permite seleccionar únicamente los casos de prueba asociados al tipo o al módulo afectado, optimizando los tiempos de ejecución.
- **Etapa de publicación de resultados:** se emplea la tarea `PublishTestResults@2` para registrar los resultados de ejecución, generar reportes y facilitar la trazabilidad a través del panel de resultados en Azure DevOps.

El diseño modular también contempla la posibilidad de reutilizar plantillas YAML para mantener consistencia entre los *pipelines*. Estas plantillas contienen la estructura básica de las tareas y permiten parametrizar variables como la ruta del proyecto, el tipo de pruebas o la rama de destino, reduciendo la necesidad de configuración manual.

### 3.3.2. Flujo de Ejecución

El flujo de ejecución del sistema propuesto se organiza en torno a tres momentos complementarios que cubren las diferentes etapas del ciclo de desarrollo de software. Cada momento responde a un objetivo distinto en cuanto a validación, cobertura y retroalimentación, buscando equilibrar la agilidad del desarrollo con la calidad continua del producto. Estos momentos son: la ejecución al crear un *Pull Request* hacia una rama de características (*feature branch*), la ejecución al integrar hacia una rama de liberación (*release branch*), y la ejecución programada nocturna (*nightly build*).

#### Momento 1: Pull Request hacia Feature Branch

En este primer momento, el pipeline se activa automáticamente cuando un desarrollador crea un *Pull Request* desde su rama de trabajo hacia una rama de características (*feature branch*). El propósito de esta ejecución es realizar una validación temprana y focalizada sobre los cambios introducidos, de modo que los errores puedan detectarse antes de su integración al flujo principal. Durante este proceso se ejecutan únicamente las pruebas unitarias asociadas a los módulos o carpetas afectadas. Esto se logra a través de filtros definidos en los archivos YAML del pipeline mediante

el parámetro `testFilterCriteria` de la tarea `VSTest@3`. De esta manera, se evita ejecutar el conjunto completo de pruebas y se obtiene retroalimentación inmediata sobre los módulos modificados, optimizando los tiempos de espera y reduciendo el uso innecesario de recursos.

#### Momento 2: Pull Request hacia Release Branch

El segundo momento ocurre cuando los cambios ya verificados en las ramas de características se integran en una rama de liberación (*release branch*). En este punto, el objetivo es validar la coherencia entre módulos y asegurar que las distintas funcionalidades convivan sin conflictos antes de una posible entrega o despliegue.
El pipeline correspondiente a esta fase ejecuta pruebas de integración y de servicio, abarcando la comunicación entre componentes, servicios o APIs. Estas pruebas garantizan que el sistema completo mantenga su estabilidad funcional y que los cambios incorporados no afecten la interacción entre módulos dependientes. En términos de ejecución, el proceso es más extenso que el anterior, pero sigue estando optimizado para ejecutarse de forma selectiva sobre los elementos realmente afectados, evitando repeticiones innecesarias en la validación.

#### Momento 3: Construcción Nocturna (Nightly Build)

Finalmente, la tercera etapa corresponde a la ejecución programada o *nightly build*, configurada mediante un *cron job* que se activa durante horas no productivas (por ejemplo, a medianoche). Esta ejecución tiene un propósito de validación global, donde se construye la solución completa y se ejecuta el conjunto total de pruebas automatizadas: unitarias, de integración y de regresión.
A diferencia de los momentos anteriores, este proceso no busca retroalimentación inmediata, sino garantizar la estabilidad general del producto a partir de las acumulaciones de cambios realizadas durante la jornada. Al ejecutarse en horarios fuera de oficina, no interfiere con la productividad de los desarrolladores y permite detectar fallos que, aunque no sean inmediatos, podrían comprometer la calidad global del sistema.

#### Síntesis del Flujo de Ejecución

La Tabla 3.3 resume los tres momentos principales de ejecución y su respectivo propósito dentro del flujo de integración continua.

| Momento | Evento que lo activa | Tipo de pruebas ejecutadas | Propósito principal |
|---|---|---|---|
| **1. PR hacia Feature Branch** | Creación de un Pull Request hacia una rama de características | Pruebas unitarias filtradas por módulo | Validar cambios tempranos y ofrecer retroalimentación inmediata. |
| **2. PR hacia Release Branch** | Integración de una rama de características en la rama de liberación | Pruebas de integración y servicio | Asegurar coherencia entre módulos antes de preparar una versión estable. |
| **3. Construcción Nocturna (Nightly Build)** | Ejecución automática mediante cron programado | Pruebas completas (unitarias, integración y regresión) | Validar la estabilidad total del sistema en horarios no productivos. |

**Tabla 3.3:** Momentos de ejecución del flujo de integración continua

Esta estructura de tres momentos permite distribuir la carga de validación de forma estratégica: las pruebas rápidas y frecuentes garantizan retroalimentación continua durante la jornada laboral, mientras que las pruebas exhaustivas se reservan para horarios de baja actividad. Así, se logra un equilibrio entre eficiencia operativa y cobertura de calidad, contribuyendo a una integración continua más fluida, confiable y alineada con las prácticas de desarrollo ágil.

## 3.4. Implementación

Esta sección describe, de forma detallada y paso a paso, el proceso técnico de implementación de la propuesta sobre Azure DevOps. El objetivo fue *reducir tiempos de ejecución y mejorar la retroalimentación* sin introducir componentes adicionales (orquestadores o infraestructura), apoyándonos únicamente en *pipelines YAML*, políticas de ramificación y la ejecución de pruebas con `VSTest@3`.

### 3.4.1. Entorno y herramientas

- **Control de versiones:** Azure Repos (Git).
- **CI:** Azure Pipelines con definiciones YAML versionadas en el repositorio.
- **Stack .NET:** SDK .NET 9.x para compilación y Runtime .NET 8.x para prueba.
- **Ejecución de pruebas:** tarea `VSTest@3` con filtrado por `Trait` (`Tipo=Unitaria`, `Tipo=Integracion`, etc.).
- **Publicación de resultados:** `PublishTestResults@2` para trazabilidad.

### 3.4.2. Estructura del repositorio

Para facilitar la segmentación por módulos y la definición de pipelines específicos, se adoptó la siguiente organización (ejemplo):

```
/
  v3/
    Samples.v3.sln
    modules/
      Modulo.A/
        src/...
        tests/Modulo.A.Tests/...
      Modulo.B/
        src/...
        tests/Modulo.B.Tests/...
      Modulo.C/...
    build/ci/
       modA.pr.yml
       modB.pr.yml
       release.yml
       nightly.yml
```

Esta estructura permitió:

1. **Encapsular** la lógica de CI por módulo (*pipelines* unitarios).

2. **Versionar** los YAML junto al código.

### 3.4.3. Estrategia de ramas y políticas

Se definieron políticas por rama para alinear los tres momentos de ejecución:

- **Feature branches** (`feature/*`): validaciones rápidas ante PR hacia `main` (unitarias de los módulos afectados).

- **Release branches** (`release/*`): pruebas de integración al proponer PR hacia `release/*`.

- **Main / release** (cron): *nightly build* ejecutando el set completo (unitarias, integración, regresión) en horario no productivo.

En Azure DevOps, esto se instrumentó con *Branch Policies* que *exigen* la ejecución exitosa de los *pipelines* correspondientes como *status checks* para completar el PR.

### 3.4.4. Etiquetado de pruebas y convenciones

Para que `VSTest@3` pueda *filtrar* por tipo y módulo sin ejecutar suites completas:

- Se usó el atributo `Trait` en las pruebas (xUnit):

```
// xUnit
[Trait("Tipo", "Unitaria")]
public class PedidoServiceTests { ... }

[Trait("Tipo", "Integracion")]
public class FacturacionApiTests { ... }
```

### 3.4.5. Pipelines por momento

#### 3.4.5.1. Momento 1: PR hacia *feature branch* (validación rápida)

En este momento se compila y ejecuta **sólo** el subconjunto mínimo (pruebas unitarias) de los módulos objetivo.

```yaml
# Deshabilita los triggers automáticos para commits directos
trigger:
  branches:
    exclude:
      - '*'

# Se ejecuta con PR hacia main
pr:
  branches:
    include:
      - main

# Build Windows + pruebas con VSTest (subset unitario)
pool:
  vmImage: 'windows-latest'

steps:
- task: UseDotNet@2
  displayName: 'Install .NET SDK 9.x'
  inputs:
    packageType: sdk
    version: '9.x'
    includePreviewVersions: false

- task: UseDotNet@2
  displayName: 'Install .NET Runtime 8.0.x'
  inputs:
    packageType: runtime
    version: '8.0.x'
```

```yaml

- script: dir $(System.DefaultWorkingDirectory) /s /b
  displayName: 'List Files in Directory ...'

- task: DotNetCoreCLI@2
  displayName: 'dotnet build'
  inputs:
    command: 'build'
    projects: 'v3/Samples.v3.sln'
    configuration: '$(buildConfiguration)'

# Ejecuta únicamente pruebas unitarias por Trait y ensamblados de interés
- task: VSTest@3
  displayName: 'Test execution (unitarias)'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: |
      **/Modulo.A.Tests.dll
      **/Modulo.B.Tests.dll
      !**/TestAdapter.dll
      !**/obj/**
    searchFolder: '$(System.DefaultWorkingDirectory)'
    testFilterCriteria: 'Tipo=Unitaria'
    codeCoverageEnabled: false

- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testRunTitle: 'PR Feature - Unit Tests'
    publishRunAttachments: true
  condition: succeededOrFailed()
  displayName: 'Publish Test Results'
```

**Archivo yml 4:** Pipeline de validación rápida para PR hacia `main` (subset unitario).

**Decisiones clave.**

- **Sin triggers por commit:** evita ejecuciones redundantes; el disparador es el PR.
- **Filtrado por Trait:** centra la ejecución en pruebas *unitarias* del módulo.

#### 3.4.5.2. Momento 2: PR hacia *release branch* (pruebas de integración)

Para garantizar *consistencia entre módulos* antes de liberar, se ejecuta un conjunto de pruebas de **integración**. Es análogo al anterior, pero con filtros `Trait` y patrones adecuados:

#### 3.4.5.3. Momento 3: *Nightly build* (cobertura completa)

Ejecución programada (cron) durante la noche, donde se corre la **batería completa** (unitarias, integración y E2E).

### 3.4.6. Buenas prácticas aplicadas

- **Consistencia de filtros:** el uso de `Trait` evita ejecutar suites ajenas al cambio.
- **Publicación uniforme de resultados:** títulos claros por momento (`PR Feature`, `PR Release`, `Nightly`) facilitan el análisis.
- **Separación por momentos:** retroalimentación *rápida* en PR de `feature`, *consistencia* en PR de `release` y *cobertura total* en el `nightly`.

.

### 3.4.7. Validación y verificación

Se ejecutaron pruebas piloto:

1. **Cambios mínimos** (p.ej., comentario en un archivo del `Modulo.A`): el pipeline de PR *sólo* ejecutó pruebas unitarias del módulo objetivo, confirmando **reducción de tiempo** frente a la ejecución completa.
2. **Cambios en múltiples módulos**: se validó que los PR a `main` disparan validaciones por módulo (según el YAML invocado por la política), y que los PR a `release/*` *ejecutan integración* con una cobertura intermedia.
3. **Nightly**: consolidó la salud del repositorio con la batería completa sin afectar horas productivas.

### 3.4.8. Lecciones y limitaciones observadas (implementación)

- *Lecciones*: La implementación del método permitió identificar varios aspectos clave que influyen directamente en la calidad y estabilidad del proceso. Uno de los aprendizajes más importantes fue que organizar el código siguiendo una estructura clara por módulos es una tarea que toma más tiempo del esperado, y que idealmente debería realizarse en etapas tempranas del desarrollo para evitar retrabajos o dependencias difíciles de rastrear. Asimismo, se evidenció que configurar los pipelines en Azure DevOps no depende únicamente del archivo YAML, sino también de las políticas y opciones que se establecen directamente en la plataforma. Esto introduce un riesgo importante: si las configuraciones no coinciden, pueden surgir errores difíciles de diagnosticar. Por esta razón, fue evidente la necesidad de actuar con precaución al ajustar tanto el código del pipeline como las reglas que lo gobiernan en Azure DevOps. Otro aprendizaje relevante fue la utilidad de definir plantillas reutilizables para los

pipelines. Contar con una base común redujo la duplicación de lógica y facilitó la creación de variantes para distintos momentos de ejecución. Este enfoque también ayudó a mantener la coherencia entre pipelines y a disminuir la posibilidad de errores manuales. Finalmente, uno de los beneficios del método propuesto es que permite establecer políticas de ejecución que no dependen del criterio del desarrollador, evitando que se omitan validaciones importantes. Esto garantiza cierta consistencia en la forma en que las pruebas se ejecutan a lo largo del proceso, incluso cuando los hábitos o estilos de trabajo del equipo son variados.

- *Limitaciones*: A pesar de los avances obtenidos, también surgieron limitaciones inherentes al enfoque utilizado. La más evidente es que el método, tal como fue aplicado, no soporta políticas para ejecutar pruebas de clases o paquetes "potencialmente afectados". Dado que la organización del proyecto se basó en carpetas por módulo, las ejecuciones se limitaron a los elementos modificados directamente, sin capacidad para identificar automáticamente dependencias indirectas que también podrían requerir validación. Además, cuando se manejan muchos módulos y diferentes tipos de prueba, configurar los pipelines manualmente puede volverse dispendioso y propenso a errores. La necesidad de crear un pipeline distinto para cada combinación de momento y conjunto de pruebas incrementa la carga operativa y exige un control más riguroso para asegurar que todos se mantengan alineados. Por último, se identificó la ausencia de un mecanismo automatizado que verifique que los pipelines fueron configurados correctamente o que su definición realmente coincide con los momentos y tipos de prueba establecidos en el diseño conceptual. Actualmente, esta validación depende de revisiones manuales, lo que abre la posibilidad de inconsistencias entre lo planificado y lo finalmente implementado.

### 3.4.9. Resultado

La implementación logró **acortar la latencia de retroalimentación** en PR de `feature` al ejecutar únicamente pruebas *unitarias* del módulo afectado, asegurar **consistencia inter-módulos** en PR de `release` con pruebas de *integración*, y mantener **cobertura total** mediante el `nightly`. Todo ello sin añadir infraestructura extra ni scripts orquestadores, y apoyándose en capacidades nativas de Azure Pipelines (`UseDotNet@2`, `DotNetCoreCLI@2`, `VSTest@3`, `PublishTestResults@2`).

## 3.5. Resumen del capítulo

El presente capítulo expuso de manera integral el proceso de diseño, desarrollo e implementación de la propuesta técnica orientada a optimizar la integración continua de pruebas automatizadas en proyectos de software gestionados con Azure DevOps. A lo largo de las secciones anteriores, se describió cómo la estructura modular, la segmentación de pruebas y la automatización de procesos permitieron reducir la redundancia en la ejecución de *pipelines* y ofrecer una retroalimentación más rápida y pertinente a los desarrolladores.

En la etapa de diseño, se establecieron los fundamentos de la solución a partir de un análisis profundo de la problemática identificada: la ejecución completa de pruebas ante cualquier cambio

en el código, incluso aquellos mínimos o triviales, que generaban sobrecarga en los servidores, tiempos de espera prolongados y desmotivación entre los equipos de desarrollo. Este diagnóstico fue el punto de partida para estructurar una propuesta basada en *pipelines YAML* modulares, configurados para ejecutar únicamente las pruebas relevantes al módulo o tipo de cambio realizado. El diseño de la solución se apoyó en la arquitectura nativa de Azure Pipelines, aprovechando su flexibilidad declarativa y las tareas estándar disponibles, tales como `DotNetCoreCLI@2` para la compilación, `VSTest@3` para la ejecución de pruebas automatizadas y `PublishTestResults@2` para la publicación de resultados. Estas tareas, al ser configurables mediante parámetros, permitieron definir una secuencia de pasos replicable, fácil de mantener y consistente entre los distintos módulos de la aplicación. Asimismo, el uso de etiquetas (`Trait`) en los métodos de prueba permitió diferenciar pruebas unitarias, de integración y de regresión, habilitando una ejecución más inteligente y focalizada.

Uno de los aportes más relevantes del capítulo fue la incorporación de la noción de "momentos" de ejecución, inspirada en el modelo CIViT y adaptada al contexto de Azure DevOps. Se establecieron tres momentos principales: (a) las validaciones rápidas ante un *Pull Request* hacia una rama de características (*feature branch*), enfocadas en pruebas unitarias; (b) las pruebas de integración ejecutadas cuando los cambios son fusionados hacia una rama de liberación (*release branch*); y (c) la construcción nocturna o *nightly build*, que ejecuta la batería completa de pruebas en horarios no productivos. Esta estrategia equilibró la velocidad y la cobertura, garantizando que cada etapa del proceso de desarrollo contara con una verificación adecuada a su nivel de impacto.

Durante la implementación, se documentaron las configuraciones técnicas, la estructura del repositorio, la política de ramas y las buenas prácticas empleadas para estandarizar la automatización. Cada *pipeline* fue implementado siguiendo un mismo patrón declarativo, lo cual no solo facilitó la trazabilidad del proceso, sino que redujo los errores humanos asociados a la configuración manual. Además, se evidenció que cuando se ejecutaba un único *pipeline* con filtros correctamente configurados, los tiempos de ejecución se reducían significativamente sin comprometer la calidad ni la integridad de los resultados.

Las pruebas realizadas confirmaron la validez de la propuesta. En los escenarios de prueba donde los cambios afectaban únicamente a un módulo, la ejecución de pruebas se limitó de forma correcta al ámbito afectado, logrando una disminución considerable en el tiempo total de retroalimentación. Por otro lado, los *Pull Requests* hacia ramas de liberación permitieron asegurar la consistencia entre módulos antes del despliegue, y las construcciones nocturnas verificaron la estabilidad global del sistema de manera automatizada.

Asimismo, durante la evaluación práctica se analizó el comportamiento del sistema cuando se modificaban simultáneamente varios módulos que contaban con sus propios *pipelines* configurados. En estos escenarios, aunque cada *pipeline* ejecutaba únicamente las pruebas correspondientes a su módulo, se evidenció un efecto colateral importante: al tratarse de procesos independientes, cada uno replicaba de forma completa la construcción de la infraestructura necesaria para compilar y ejecutar las pruebas. Esto implicó que, ante cambios que afectaban tres módulos, por ejemplo, se disparaban tres *pipelines* consecutivos o en cascada, cada uno repitiendo el mismo proceso de inicialización, descarga de dependencias y preparación del entorno.

Si bien la ejecución de las pruebas en sí misma fue más rápida gracias al filtrado por módulos, el tiempo total de retroalimentación se vio afectado por la duplicación o triplicación del tiempo invertido en la fase de construcción. Este hallazgo permitió identificar una oportunidad clara de mejora para futuros trabajos: consolidar la construcción de infraestructura en un único proceso centralizado o compartido, que permita obtener los beneficios de la segmentación sin incurrir en el costo adicional de repetir operaciones idénticas en múltiples *pipelines*.

Finalmente, este capítulo deja sentadas las bases para la siguiente etapa del trabajo: la evaluación formal de la solución y la identificación de métricas de desempeño como la latencia de retroalimentación, la cobertura de pruebas efectivas y la tasa de fallos detectados por cada momento de ejecución. Estas métricas permitirán cuantificar de forma objetiva el impacto de la propuesta, demostrando su aporte a la mejora continua del proceso de desarrollo de software en entornos ágiles y colaborativos.

En síntesis, la implementación desarrollada consolidó un modelo de integración continua más ágil, inteligente y alineado con las necesidades reales del equipo de desarrollo. La combinación de una arquitectura modular, políticas de ejecución por momento y uso eficiente de herramientas nativas de Azure DevOps permitió construir un proceso que equilibra la velocidad de entrega, la confiabilidad del código y la eficiencia operativa. Este enfoque no solo responde a la problemática inicial planteada, sino que también sienta las bases para la evolución futura hacia esquemas más avanzados de automatización y orquestación dentro de la cultura DevOps.

Capítulo 4

# Evaluación

## 4.1. Diseño de la Evaluación

La evaluación de la propuesta se realizó con el propósito de analizar su eficacia en términos de tiempo de retroalimentación, pertinencia de ejecución de pruebas y percepción del usuario final dentro del contexto real de uso. Esta fase tuvo un enfoque cualitativo, orientado a comprender el impacto práctico de la solución y a validar si las condiciones operativas observadas coinciden con los objetivos y beneficios teóricos planteados durante el diseño del proceso de retroalimentación oportuna.

### 4.1.1. Enfoque metodológico

Dado que el proyecto se centra en un proceso técnico dentro de un entorno corporativo real, la evaluación adoptó un enfoque cualitativo basado en entrevista semiestructurada y observación participante. La elección de este enfoque se sustentó en la necesidad de capturar percepciones, experiencias y valoraciones subjetivas del usuario experto —en este caso, el jefe del área de DevOps—, cuya perspectiva permite validar la aplicabilidad, mantenibilidad y eficiencia de la solución más allá de las métricas cuantitativas.
El método cualitativo resulta especialmente útil en escenarios donde los resultados técnicos dependen de la interacción humana con las herramientas, de los hábitos de los equipos de trabajo y de la estructura organizacional. Por ello, se priorizó comprender **cómo la implementación impacta la rutina diaria del equipo de desarrollo y los tiempos de validación de código**, más que únicamente medir tiempos o porcentajes de ejecución.
La entrevista se diseñó en torno a tres ejes de análisis:

1. **Eficiencia operativa:** percepción del cambio en tiempos de ejecución, carga de trabajo y consumo de recursos al aplicar la nueva estructura de pipelines.

2. **Calidad y oportunidad de la retroalimentación:** percepción sobre la utilidad de recibir resultados segmentados por módulo y tipo de cambio.

3. **Mantenibilidad y complejidad de configuración:** valoración sobre la facilidad o dificultad para crear y mantener los archivos YAML y las políticas de ejecución.

Cada uno de estos ejes permitió analizar los efectos del sistema en las tres dimensiones más relevantes para una solución DevOps: *tiempo*, *calidad* y *operatividad*.

### 4.1.2. Participante y contexto organizacional

El participante principal de la evaluación fue el jefe del área de DevOps de la organización colaboradora. Su rol dentro de la empresa implica la supervisión de las políticas de integración y despliegue, así como la configuración de los entornos de pruebas y la gestión de la infraestructura asociada a los pipelines.
El contexto de la evaluación se desarrolló dentro de un entorno corporativo real, donde los procesos de integración continua ya se encontraban implementados con prácticas estandarizadas. La empresa posee una estructura modular en su base de código, con múltiples proyectos agrupados bajo una misma solución (.NET), y mantenía hasta antes de la propuesta un esquema tradicional de ejecución completa de pruebas ante cualquier modificación del código. Esta práctica generaba latencias considerables en la entrega de resultados y afectaba directamente la agilidad del equipo de desarrollo, especialmente durante etapas críticas de liberación de producto.
La prueba piloto del nuevo modelo se aplicó sobre tres módulos específicos del sistema, seleccionados por su nivel de actividad y frecuencia de cambios en el repositorio. Cada módulo contaba con un pipeline independiente, parametrizado para ejecutar únicamente las pruebas relevantes según el tipo de cambio. Este entorno de validación permitió observar de manera precisa los efectos positivos y negativos del nuevo enfoque.

### 4.1.3. Diseño del instrumento de evaluación

El instrumento de evaluación consistió en una entrevista semiestructurada aplicada al participante principal. Se diseñaron diez preguntas abiertas que permitieron explorar la percepción del usuario en torno a tres dimensiones de análisis: desempeño, experiencia operativa y alineación con los objetivos organizacionales.

**Tabla 4.1:** Estructura del instrumento de entrevista

| **Dimensión** | **Preguntas guía de la entrevista** |
|---|---|
| Eficiencia operativa | ¿Percibe una reducción en los tiempos de ejecución de los pipelines?<br>¿En qué casos considera que el nuevo modelo aporta más valor (módulos pequeños, grandes o múltiples)? |
| Retroalimentación y calidad | ¿Cómo califica la oportunidad de la retroalimentación obtenida en los tres momentos de ejecución (PR feature, PR release, nightly)?<br>¿Considera que la segmentación de pruebas mejora la precisión del feedback técnico? |
| Mantenibilidad y configuración | ¿Qué nivel de dificultad encuentra al crear y mantener los archivos YAML?<br>¿Existen riesgos o errores recurrentes en la configuración manual de los pipelines? |
| Alineación con objetivos de negocio | ¿De qué manera cree que la nueva estructura contribuye a la productividad del equipo?<br>¿Considera que esta metodología es sostenible a largo plazo dentro de la empresa? |

Las entrevistas fueron registradas y posteriormente analizadas bajo el enfoque de *codificación temática*, identificando patrones y relaciones entre las respuestas. Esta técnica permitió extraer las conclusiones más relevantes y estructurar las observaciones en categorías que se alinean con los objetivos específicos del trabajo.

### 4.1.4. Procedimiento de evaluación

El proceso de evaluación se desarrolló en tres etapas consecutivas:

1. **Preparación:** se configuraron los tres pipelines de prueba (uno por módulo), asegurando que los parámetros de filtrado y las políticas de activación coincidieran con los tres momentos definidos en el diseño de la propuesta: validación rápida, integración y ejecución completa nocturna.

2. **Aplicación:** se simularon escenarios de cambio controlados, abarcando tanto modificaciones menores (como un cambio en un archivo de código o comentario), como modificaciones mayores (que afectaban múltiples módulos en paralelo). Durante esta fase, se ejecutaron los pipelines y se documentaron los tiempos de respuesta y los resultados obtenidos.

3. **Entrevista:** finalmente, se llevó a cabo la entrevista con el jefe de DevOps, quien analizó los resultados y aportó una evaluación cualitativa sobre la utilidad y las limitaciones del sistema desde su experiencia práctica.

El análisis cualitativo permitió complementar la observación técnica con una valoración profesional, identificando los escenarios en los que la propuesta aporta beneficios tangibles y aquellos donde genera efectos contraproducentes.

### 4.1.5. Criterios de evaluación

Para estructurar la interpretación de los resultados, se definieron tres criterios de evaluación cualitativa:

- **Eficiencia:** nivel en que la propuesta reduce el tiempo total de ejecución de pruebas y mejora la velocidad de retroalimentación sin comprometer la calidad.
- **Eficacia:** grado en que la propuesta logra ejecutar únicamente las pruebas pertinentes y relevantes a los cambios realizados.
- **Usabilidad operativa:** facilidad de implementación, mantenibilidad y sostenibilidad del modelo dentro del flujo de trabajo del equipo de DevOps.

Estos criterios se analizaron no con métricas numéricas, sino mediante una valoración cualitativa derivada de la experiencia y percepción del usuario experto, respaldada por observaciones documentadas durante la fase de prueba.

### 4.1.6. Limitaciones del diseño de evaluación

Aunque el método cualitativo permitió obtener una visión profunda del impacto humano y técnico de la propuesta, también presenta limitaciones inherentes. En primer lugar, la evaluación se centró en un único participante experto, lo que restringe la diversidad de perspectivas. Además, al realizarse en un entorno de prueba controlado y no durante un ciclo completo de desarrollo, algunos comportamientos asociados a la carga real de trabajo podrían no haberse manifestado con exactitud.

No obstante, el valor de este tipo de evaluación radica en su capacidad para identificar los **puntos críticos de mejora**, las **percepciones de valor** y las **implicaciones prácticas** del sistema dentro de su contexto real. En este sentido, los hallazgos cualitativos obtenidos aportan una base sólida para la toma de decisiones en futuras iteraciones del modelo.

### 4.1.7. Síntesis del diseño de evaluación

En síntesis, el diseño de la evaluación cualitativa permitió contrastar la teoría con la práctica, validando los efectos reales del modelo de integración continua propuesto. El uso de la entrevista como instrumento central posibilitó una comprensión holística de los beneficios y desafíos del sistema desde la perspectiva operativa de DevOps.

El enfoque adoptado no buscó únicamente medir tiempos o indicadores cuantitativos, sino comprender la experiencia del usuario, la interacción entre componentes técnicos y el grado de alineación entre los objetivos del sistema y las necesidades reales de la organización. Este análisis cualitativo

se convierte, por tanto, en un insumo fundamental para la interpretación de los resultados que se presentan en la siguiente sección.

## 4.2. Resultados de la evaluación

Esta sección presenta los resultados obtenidos tras la aplicación de la evaluación cualitativa del proyecto, realizada mediante entrevistas, observación y revisión de las ejecuciones del sistema en un entorno real de trabajo. El propósito fue analizar el impacto de la propuesta sobre el flujo de integración continua, la eficiencia en la ejecución de pruebas automatizadas y la percepción del responsable del área de DevOps frente a la utilidad, mantenibilidad y efectividad de la solución implementada.

### 4.2.1. Análisis general de la entrevista

El análisis de las respuestas del jefe de DevOps permitió identificar una percepción general positiva respecto al enfoque modular y al filtrado de pruebas por tipo y módulo. Sin embargo, también se evidenciaron limitaciones prácticas que afectan parcialmente la eficiencia esperada cuando los cambios abarcan múltiples módulos del sistema.
El entrevistado destacó que, en los casos donde la modificación se realiza sobre un único módulo, la mejora es inmediata: el pipeline ejecuta únicamente las pruebas correspondientes, reduciendo el tiempo total de ejecución y liberando recursos para otros procesos. En cambio, cuando las modificaciones afectan varios módulos interrelacionados, se observa un comportamiento diferente: cada pipeline individual inicia su propio proceso de construcción (*build*), lo cual incrementa significativamente el tiempo total de ejecución y el consumo de recursos.
En palabras del entrevistado, *"la solución es muy efectiva cuando se trabaja en un módulo a la vez, pero se vuelve costosa en términos de tiempo cuando hay múltiples pipelines ejecutando su propia construcción en paralelo"*. Esta observación refleja un punto crítico que evidencia la necesidad de avanzar hacia un esquema más inteligente de orquestación, donde la infraestructura se construya una sola vez y las pruebas se distribuyan de forma condicional entre los módulos correspondientes.

### 4.2.2. Resultados por criterio de evaluación

A continuación, se presentan los hallazgos organizados en función de los criterios definidos en el diseño de la evaluación (eficiencia, eficacia y usabilidad operativa). Cada criterio incluye una síntesis de los resultados observados, la valoración cualitativa del usuario y las implicaciones para la propuesta.

**Tabla 4.2:** Resultados cualitativos por criterio de evaluación

| Criterio | Hallazgos principales | Valoración del usuario |
|---|---|---|
| **Eficiencia** | La ejecución de pruebas por módulo redujo el tiempo promedio de retroalimentación en un 60 % cuando se modificó un único módulo. Sin embargo, la ejecución simultánea de múltiples pipelines genera sobrecarga de construcción y disminuye la ganancia de eficiencia esperada. | Alta (para un módulo) Media-baja (para múltiples módulos). |
| **Eficacia** | El filtrado de pruebas mediante `Trait` se comportó correctamente, ejecutando solo los casos relevantes. No se detectaron ejecuciones redundantes en las pruebas unitarias. Sin embargo, el impacto del tiempo de construcción limita la percepción general de mejora. | Alta, con observaciones sobre el orden de ejecución de etapas. |
| **Usabilidad operativa** | La creación manual de archivos YAML por módulo resultó funcional pero repetitiva. El entrevistado resaltó el riesgo de errores humanos y la falta de una herramienta que automatice la generación o validación de estos archivos. | Media, con sugerencia de automatización mediante plantillas o scripts validados. |

De acuerdo con la tabla anterior, se puede concluir que la propuesta cumple satisfactoriamente con los objetivos de segmentación y filtrado de pruebas, pero requiere optimización en la etapa de construcción (*build*) cuando múltiples módulos son procesados en paralelo. Esta situación no invalida el modelo, sino que enfatiza su necesidad de evolución hacia un sistema de ejecución inteligente y centralizado.

### 4.2.3. Percepción sobre los momentos de ejecución

La implementación se evaluó en los tres momentos definidos en el diseño de la propuesta: *Pull Request* hacia una *feature branch*, *Pull Request* hacia una *release branch*, y la ejecución programada nocturna (*nightly build*). La Tabla 4.3 resume los resultados observados y la percepción del usuario para cada momento.

**Tabla 4.3:** Percepción del usuario por momento de ejecución

| Momento de ejecución | Resultado observado | Percepción del jefe de DevOps |
|---|---|---|
| **PR hacia feature branch** | Ejecución rápida y focalizada. La reducción del tiempo de retroalimentación fue notable (hasta 70 %). | Alta: percibida como una mejora significativa para desarrolladores. |
| **PR hacia release branch** | Validación adecuada de integración entre módulos, pero con incremento de tiempos debido a la reconstrucción total de la solución en cada pipeline. | Media: útil para garantizar estabilidad, pero con alto costo de ejecución. |
| **Nightly build** | Ejecución completa, estable y controlada. Se ejecutan todas las pruebas sin interferir con las horas laborales. | Alta: percibida como una práctica de calidad continua altamente beneficiosa. |

Los resultados confirman que el modelo logra una relación equilibrada entre cobertura y retroalimentación oportuna. Los pipelines asociados a *feature branches* son los más eficientes en tiempo, mientras que las ejecuciones programadas (nightly) garantizan una validación integral del sistema. Sin embargo, los pipelines de integración (*release*) requieren una optimización adicional para reducir la redundancia en las tareas de construcción.

### 4.2.4. Análisis cualitativo de ventajas y desventajas observadas

Durante la entrevista y la observación técnica, se identificaron las principales ventajas y desventajas de la solución implementada. Estas se resumen en la Tabla 4.4.

**Tabla 4.4:** Ventajas y desventajas identificadas durante la evaluación

| Ventajas | Desventajas / Limitaciones |
|---|---|
| Ejecuta solo las pruebas necesarias, evitando redundancia en los módulos no modificados. | Cuando se modifican múltiples módulos, cada pipeline construye su propia infraestructura, incrementando el tiempo total de ejecución. |
| Aumenta la velocidad de retroalimentación para cambios pequeños y pruebas unitarias. | No existe un mecanismo de coordinación entre pipelines que evite duplicaciones de construcción. |
| Facilita la trazabilidad y el análisis de resultados gracias a la segmentación por módulo. | Requiere creación manual de archivos YAML, lo cual puede generar errores humanos en entornos con múltiples módulos. |
| Promueve una cultura de calidad continua y control de cambios en tiempo real. | El mantenimiento manual de políticas de ejecución se vuelve complejo a medida que el proyecto crece. |

El balance general de la evaluación demuestra que la propuesta resuelve el problema inicial de retroalimentación tardía y sobrecarga de pruebas, aunque revela oportunidades claras de mejora en la automatización y en la gestión de dependencias entre pipelines.

### 4.2.5. Discusión de los resultados

Desde una perspectiva operativa, los hallazgos evidencian que la propuesta logra cumplir su propósito principal en escenarios controlados o de baja concurrencia, donde los cambios se limitan a un único módulo. En estos casos, los beneficios son tangibles: la retroalimentación se recibe en menos tiempo, las pruebas son específicas y el consumo de recursos se mantiene bajo.
Sin embargo, al escalar la solución a escenarios donde múltiples módulos son modificados, la ventaja inicial se ve atenuada por el costo acumulado de construcción. Este hallazgo confirma una de las hipótesis planteadas en el diseño del proyecto: que el punto crítico del proceso no se encuentra en la ejecución de pruebas, sino en la reconstrucción repetida del entorno. Por tanto, los resultados de esta evaluación cualitativa sustentan la necesidad de incorporar en el futuro un mecanismo de orquestación centralizado o una etapa compartida de construcción para mejorar la eficiencia global del proceso.
Otro hallazgo relevante fue la valoración del participante sobre la **facilidad de adopción** de la metodología. A pesar de requerir configuraciones manuales iniciales, el entrevistado resaltó que la lógica es fácilmente comprensible para los equipos técnicos y que la documentación adjunta facilita la capacitación de nuevos miembros. Este factor refuerza la aplicabilidad práctica de la propuesta en entornos empresariales donde la curva de aprendizaje y la mantenibilidad son factores determinantes para la adopción de nuevas herramientas.
En términos generales, la evaluación permitió validar que el sistema cumple con los objetivos específicos de la investigación, especialmente en lo que respecta a la optimización del flujo de pruebas y a la entrega de retroalimentación oportuna. No obstante, también identificó las condiciones necesarias para su perfeccionamiento, entre ellas: la consolidación de pipelines, la automatización de configuraciones YAML y la integración de un sistema de detección de dependencias afectadas.

### 4.2.6. Conclusión de los resultados

Los resultados de esta evaluación demuestran que la propuesta implementada ofrece mejoras significativas en la eficiencia y pertinencia del proceso de integración continua. El enfoque modular y la segmentación de pruebas se traducen en una reducción considerable del tiempo de retroalimentación, especialmente en los casos de cambios unitarios o localizados.
Sin embargo, la falta de un mecanismo unificado de construcción limita parcialmente la eficiencia global cuando múltiples módulos son modificados simultáneamente. Esta situación representa una oportunidad concreta de mejora que ya se ha proyectado como parte de los trabajos futuros descritos en el Capítulo 5.
En conclusión, la evaluación cualitativa permitió validar empíricamente que la solución es viable, útil y coherente con las necesidades reales del entorno DevOps, aportando una base sólida para evolucionar hacia una automatización más integral y eficiente.

## 4.3. Resumen del capítulo

El presente capítulo abordó el proceso de evaluación de la propuesta desarrollada, cuyo objetivo fue analizar de manera crítica y sistemática el impacto de la solución sobre el flujo de integración continua en entornos reales de trabajo. Esta evaluación, de naturaleza cualitativa, permitió comprender en profundidad no solo los resultados técnicos de la implementación, sino también la percepción y experiencia del usuario experto a cargo del área de DevOps, consolidando una visión integral sobre la aplicabilidad, beneficios y limitaciones del modelo propuesto.

### 4.3.1. Síntesis metodológica

La evaluación se estructuró con base en un enfoque cualitativo de tipo exploratorio-aplicado, centrado en la observación y la entrevista semiestructurada. Esta metodología permitió capturar evidencias desde la práctica operativa diaria, favoreciendo un análisis más cercano a la realidad organizacional, donde los factores humanos, técnicos y de gestión interactúan constantemente.
A diferencia de un enfoque cuantitativo tradicional, que busca validar hipótesis mediante mediciones estadísticas, este modelo de evaluación priorizó la *comprensión contextual*, es decir, cómo la solución influye en los procesos reales y en la experiencia del usuario. De este modo, se logró identificar con claridad las áreas de mejora que no habrían sido visibles a través de métricas puramente numéricas, como el impacto emocional de la espera de resultados, la percepción de eficiencia por parte de los desarrolladores o la carga cognitiva asociada a la configuración manual de pipelines.
El instrumento de recolección de datos, basado en una entrevista dirigida al jefe del área de DevOps, abordó tres ejes fundamentales: la eficiencia del proceso, la calidad y oportunidad de la retroalimentación, y la usabilidad operativa del modelo. Estos ejes proporcionaron una estructura coherente que permitió comparar la experiencia antes y después de la implementación, así como identificar los factores que más influyen en la percepción de valor del nuevo flujo de trabajo.

### 4.3.2. Principales hallazgos y análisis interpretativo

Los resultados de la evaluación evidenciaron una mejora significativa en los escenarios donde el modelo propuesto se aplica sobre un único módulo del sistema. En estos casos, el pipeline ejecuta únicamente las pruebas relevantes, evitando la ejecución completa de la batería de pruebas automatizadas. Este comportamiento se tradujo en una reducción del tiempo promedio de retroalimentación cercana al 60 %, permitiendo que los desarrolladores recibieran resultados en cuestión de minutos en lugar de esperar procesos de hasta media hora.
Desde la perspectiva del usuario evaluador, esta reducción en los tiempos de espera constituye una de las mejoras más notables del nuevo modelo. El entrevistado enfatizó que los desarrolladores sienten mayor control sobre su flujo de trabajo, dado que la retroalimentación llega de forma más rápida y precisa, lo cual contribuye directamente a la productividad del equipo y a la detección temprana de errores.
Sin embargo, la evaluación también permitió identificar un punto crítico de mejora: cuando se modifican varios módulos de manera simultánea, el sistema ejecuta múltiples pipelines en paralelo,

cada uno con su propia etapa de construcción. Este comportamiento, aunque funcional, genera un incremento en los tiempos totales de ejecución, dado que la infraestructura se reconstruye tantas veces como pipelines existan en ejecución. En este sentido, se observó que el mayor consumo de tiempo no se encuentra en las pruebas, sino en la fase de construcción (*build*), la cual resulta redundante entre pipelines.
Este hallazgo representa una evidencia clave para la siguiente etapa de optimización del sistema: la necesidad de diseñar un **pipeline orquestador centralizado** que construya la infraestructura una sola vez y ejecute condicionalmente las pruebas de cada módulo según los cambios detectados. La evaluación cualitativa permitió, por tanto, no solo validar el funcionamiento del modelo actual, sino también señalar el camino lógico hacia su evolución técnica.
Otro resultado relevante se relaciona con la **usabilidad operativa**. El jefe de DevOps manifestó que la creación manual de archivos YAML por módulo, aunque viable, introduce riesgos de errores humanos, especialmente cuando el número de módulos supera las dos decenas. Este problema se intensifica en entornos donde varios ingenieros participan simultáneamente en la configuración y mantenimiento de pipelines, pues la inconsistencia entre archivos o la omisión de parámetros específicos puede provocar ejecuciones fallidas o comportamientos inesperados.
La experiencia del usuario también reveló un aspecto positivo: la claridad del diseño. El modelo conceptual fue percibido como intuitivo, fácil de documentar y comprensible incluso para ingenieros que no están familiarizados con Azure DevOps. La segmentación del proceso en tres momentos (*Pull Request Feature*, *Pull Request Release* y *Nightly Build*) resultó especialmente útil para la trazabilidad del proceso y la organización del trabajo. Este enfoque refleja una mejora sustancial respecto a la ejecución monolítica de pruebas que predominaba antes de la implementación.

### 4.3.3. Interpretación técnica de los resultados

Desde el punto de vista técnico, la evaluación permitió confirmar que los objetivos específicos de la investigación fueron alcanzados. La propuesta demostró ser capaz de:

- Ejecutar únicamente las pruebas relevantes según los cambios realizados en el código fuente.
- Reducir la redundancia en la ejecución de pruebas automatizadas.
- Mejorar los tiempos de retroalimentación para el desarrollador.
- Mantener la consistencia en la calidad de los resultados de las pruebas.

No obstante, se identificaron limitaciones relacionadas con la capacidad del sistema para manejar múltiples módulos de manera simultánea. Al ejecutar más de un pipeline en paralelo, los beneficios iniciales de reducción de tiempo se ven parcialmente anulados por la sobrecarga de construcción repetida. Esta situación pone de manifiesto la necesidad de incorporar una *capa de inteligencia operacional* que permita detectar qué módulos comparten dependencias y coordinar una única etapa de construcción para todos ellos.
El análisis también evidenció la importancia de fortalecer la automatización del proceso de configuración. Actualmente, los archivos YAML son creados y mantenidos manualmente, lo que aumenta la

probabilidad de errores por omisión, duplicidad o inconsistencias en los parámetros definidos. Una posible mejora a corto plazo sería la generación automática de estos archivos a partir de plantillas controladas o scripts PowerShell validados, lo cual reduciría el margen de error y simplificaría la adopción de nuevas configuraciones.

### 4.3.4. Implicaciones prácticas y organizacionales

A nivel organizacional, los resultados de la evaluación demuestran que la propuesta genera un impacto positivo en la productividad y en la gestión del tiempo de los equipos de desarrollo. La segmentación del proceso de integración en distintos momentos permite distribuir la carga de trabajo de forma más equilibrada y predecible, lo cual contribuye a reducir la presión sobre los equipos de QA y desarrollo durante las fases críticas de liberación de producto.
Asimismo, el modelo refuerza la cultura de mejora continua dentro del área de DevOps, fomentando la revisión constante de las prácticas de automatización y la búsqueda de optimización de los procesos internos. Este tipo de beneficios, aunque difíciles de medir cuantitativamente, poseen un alto valor estratégico para las organizaciones tecnológicas, pues impactan directamente en la eficiencia operativa y en la satisfacción del personal técnico.
Por otra parte, la propuesta evidencia la relevancia de contar con una infraestructura flexible y escalable. Si bien el modelo actual cumple con los objetivos iniciales, la empresa requiere una evolución que permita absorber la creciente complejidad de sus proyectos. En este sentido, el modelo CIViT adaptado a Azure DevOps y los principios de segmentación de pruebas constituyen una base sólida sobre la cual construir versiones más robustas y automatizadas del sistema.

### 4.3.5. Lecciones aprendidas derivadas del proceso de evaluación

Durante la ejecución del proyecto y la aplicación del modelo, se obtuvieron varias lecciones relevantes que complementan los hallazgos de la evaluación:

- La organización temprana del código por módulos facilita la integración del modelo de pruebas, pero exige disciplina y tiempo de planificación inicial.
- Configurar correctamente los pipelines requiere comprender tanto el código YAML como las políticas de ejecución de Azure DevOps. La inconsistencia entre ambos puede generar fallos difíciles de detectar.
- El uso de plantillas reutilizables representa una buena práctica para garantizar la homogeneidad en la configuración de múltiples pipelines.
- La definición de políticas automáticas reduce la dependencia del criterio del desarrollador al momento de ejecutar pruebas, asegurando la consistencia del proceso.

Estas lecciones evidencian que el éxito de la propuesta no depende exclusivamente de la tecnología empleada, sino también de la calidad del diseño organizacional y de la capacidad del equipo para adoptar nuevas prácticas con una visión de mejora continua.

### 4.3.6. Conclusión del capítulo

El proceso de evaluación cualitativa permitió validar la viabilidad técnica y operativa del modelo de integración continua propuesto, al tiempo que reveló áreas claras de mejora. Los resultados confirman que la solución contribuye a resolver el problema original de retroalimentación tardía, optimizando los tiempos de respuesta y mejorando la calidad de los procesos de integración y prueba.

Al mismo tiempo, la evaluación puso de manifiesto que el sistema actual puede evolucionar hacia una versión más inteligente y automatizada, capaz de gestionar de forma eficiente las dependencias entre módulos y de minimizar el tiempo de construcción global. Esta conclusión refuerza la relevancia del modelo propuesto como base para futuras investigaciones en automatización de pipelines, orquestación condicional y estrategias de testing adaptativo.

En síntesis, la evaluación constituyó no solo un ejercicio de validación técnica, sino también una reflexión crítica sobre los procesos de integración continua en entornos empresariales reales, aportando conocimiento práctico y teórico que puede servir como referencia para proyectos futuros en el campo de la ingeniería de software y la automatización DevOps.

Finalmente, como parte del cierre de este capítulo, se deja constancia de que las evidencias recolectadas durante la evaluación —incluyendo la encuesta aplicada al jefe del área de DevOps, los registros de ejecución del pipeline, y las capturas de pantalla de los resultados obtenidos— se encuentran compiladas en el Capítulo 6: Anexos. Dichas evidencias complementan el análisis presentado en este apartado y respaldan los hallazgos descritos, sirviendo como material de referencia para futuras revisiones o validaciones del trabajo desarrollado

Capítulo 5

# Conclusiones

Este capítulo presenta las conclusiones finales del trabajo de investigación, las proyecciones a futuro y las lecciones aprendidas durante la ejecución del proyecto. Se responde a los objetivos planteados, se analizan los resultados obtenidos frente a la literatura existente y se reconocen las limitaciones y oportunidades de mejora que surgieron durante la implementación de la propuesta.

El propósito principal de esta investigación fue proponer un método de diseño de procesos de integración continua que permitiera organizar conjuntos de pruebas automatizadas y ofrecer retroalimentación oportuna a los desarrolladores. Este objetivo se cumplió mediante la implementación de una arquitectura basada en *pipelines YAML* en Azure DevOps, la configuración de políticas de ejecución asociadas a ramas específicas y la aplicación de criterios de filtrado de pruebas mediante el uso del atributo `Trait` en los frameworks de pruebas.

Los resultados demostraron que, al ejecutar únicamente las pruebas asociadas al módulo modificado, los tiempos de retroalimentación se redujeron significativamente frente al enfoque tradicional donde todas las pruebas del sistema se ejecutaban sin distinción. La separación del proceso en tres momentos de validación —*Pull Request* hacia *feature branch*, *Pull Request* hacia *release branch* y construcción nocturna (*nightly build*)— permitió equilibrar cobertura, eficiencia y disponibilidad de recursos. Con ello se logró una automatización más inteligente y adaptable, coherente con las prácticas modernas de *DevOps* y los principios del modelo *CIViT* adaptado.

En comparación con la literatura existente, los resultados de esta investigación coinciden con los hallazgos de (Machalica et al., 2019) y (Mehta et al., 2021a), quienes evidenciaron que la priorización y selección de pruebas a partir de criterios específicos contribuye a la reducción de costos y tiempos de ejecución. De igual manera, los estudios de (Elsner et al., 2022) y (Fallahzadeh et al., 2023) respaldan la idea de que la optimización del flujo de integración incrementa la productividad y disminuye la latencia de retroalimentación. La propuesta de este trabajo aporta un enfoque pragmático que traduce dichos principios teóricos en una solución operativa, accesible y reproducible dentro de un entorno empresarial.

Desde una perspectiva práctica, esta investigación aporta una metodología replicable para organizaciones que deseen optimizar sus procesos de integración continua sin necesidad de introducir herramientas adicionales o infraestructura compleja. Al apoyarse exclusivamente en las capacidades nativas de Azure DevOps, la propuesta demuestra que es posible alcanzar una retroalimentación eficiente, modular y controlada con bajo costo de adopción y mantenimiento.

En síntesis, el proyecto alcanzó los siguientes logros:

- Reducción del tiempo promedio de retroalimentación en más del 60 % al ejecutar solo pruebas relevantes por módulo.

- Incremento en la trazabilidad de los resultados gracias a la estandarización de tareas y nombres de ejecución.
- Implementación de una estructura modular que facilita la escalabilidad del proceso de integración continua.
- Adopción de prácticas que alinean la automatización con las políticas de calidad del código y las reglas de revisión de *Pull Requests*.

## 5.1. Trabajos Futuros

Durante el desarrollo del proyecto surgieron oportunidades de mejora orientadas a profundizar y expandir los alcances de esta investigación. Entre las principales líneas de trabajo futuro se destacan:

1. **Automatización de la detección de cambios:** implementar scripts o funciones que identifiquen automáticamente los módulos o archivos modificados y ajusten dinámicamente los parámetros del *pipeline* para ejecutar solo las pruebas correspondientes.
2. **Creación de un pipeline orquestador:** consolidar la ejecución de los distintos *pipelines* en un flujo único que determine condicionalmente qué tareas deben ejecutarse, evitando configuraciones repetitivas y reduciendo el mantenimiento operativo.
3. **Integración de métricas de rendimiento:** desarrollar un tablero de métricas que mida la latencia de retroalimentación, la cobertura efectiva de pruebas, la estabilidad del *build* y la tasa de falsos positivos.
4. **Ampliación del alcance de pruebas:** incorporar pruebas de aceptación y de interfaz de usuario automatizadas en el flujo, integrando herramientas como Playwright o Selenium.
5. **Detección de dependencias afectadas:** explorar algoritmos o heurísticas que permitan identificar automáticamente las clases o paquetes potencialmente impactados por un cambio, extendiendo así la cobertura del modelo actual.

Estos trabajos futuros buscan consolidar una segunda versión de la propuesta, en la cual la automatización y la adaptabilidad permitan alcanzar un proceso de integración continua completamente autónomo, eficiente y alineado con los principios de calidad continua en entornos de *DevOps*.

## 5.2. Lecciones Aprendidas

Durante la ejecución del proyecto se identificaron aprendizajes clave que aportaron tanto al resultado técnico como al desarrollo profesional del autor. Estas lecciones reflejan la madurez adquirida en la gestión de proyectos de automatización y la importancia de la planificación temprana en entornos de integración continua.

**Tabla 5.1:** Principales Lecciones Aprendidas

| Lección | Descripción |
|---|---|
| Organización temprana del código | Organizar el código por módulos desde las etapas iniciales reduce la complejidad en la configuración y mantenimiento de los pipelines. |
| Configuración dual en Azure DevOps | Configurar los pipelines tanto en el YAML como en las políticas de ejecución requiere coherencia; cualquier desajuste puede generar errores difíciles de rastrear. |
| Uso de plantillas reutilizables | La creación de plantillas YAML reutilizables simplifica el desarrollo y mantiene la coherencia en los distintos pipelines. |
| Automatización de políticas | Definir políticas automáticas garantiza que las reglas de calidad se apliquen consistentemente, sin depender de la acción manual del desarrollador. |
| Gestión de tiempos de compilación y prueba | Comprender las etapas que más consumen tiempo (compilación y pruebas) permitió identificar puntos críticos para optimización. |

Estas lecciones reafirman la importancia de la estandarización, la modularidad y la planificación anticipada en la gestión de procesos automatizados.

## 5.3. Limitaciones del Proyecto

A pesar de los logros alcanzados, la investigación presentó ciertas limitaciones que abren nuevas oportunidades de mejora y desarrollo:

- El modelo actual no contempla la ejecución de pruebas sobre clases o paquetes potencialmente afectados por cambios indirectos; su enfoque se limita a módulos directamente modificados.
- La configuración manual de múltiples *pipelines* puede resultar laboriosa y propensa a errores cuando el número de módulos y tipos de prueba crece.
- No se cuenta con un mecanismo automatizado que valide si las configuraciones de los *pipelines* coinciden con la definición teórica de los “momentos” de ejecución.

Estas limitaciones no afectan la validez del modelo propuesto, pero señalan la necesidad de incorporar mecanismos de análisis y verificación automática en futuras versiones del sistema.

## 5.4. Resumen del Capítulo

En conclusión, este trabajo logró diseñar e implementar un método de integración continua más eficiente, segmentado y alineado con las necesidades reales del proceso de desarrollo de software.

La propuesta contribuyó a resolver el problema identificado de retroalimentación tardía, implementando una solución modular que aprovecha las capacidades nativas de Azure DevOps y adopta un enfoque de ejecución por momentos basado en el modelo *CIViT*.
Los resultados obtenidos evidencian que la automatización estratégica de pruebas no solo mejora la eficiencia técnica, sino también la cultura organizacional al fomentar prácticas de desarrollo más disciplinadas, colaborativas y sostenibles. A su vez, el conocimiento adquirido y las lecciones aprendidas fortalecen la capacidad del autor para enfrentar nuevos retos en el ámbito de la automatización, la ingeniería de software y la integración continua.
Finalmente, las limitaciones y trabajos futuros delinean un camino claro hacia la evolución del modelo, orientado a alcanzar una automatización completa y adaptativa. Este proyecto deja como legado una base metodológica sólida y una implementación funcional que puede servir como referencia para equipos y organizaciones que busquen optimizar su flujo de desarrollo bajo los principios de calidad, eficiencia y mejora continua.

Capítulo 6

# Anexos

El presente capítulo reúne la documentación y los soportes visuales que complementan el desarrollo del proyecto. Su propósito es ofrecer un respaldo tangible a los resultados descritos en los capítulos anteriores, especialmente en la sección de evaluación, donde se analizaron los impactos técnicos y operativos de la implementación propuesta. Aquí se incluyen los pantallazos, evidencias y registros asociados al proceso de evaluación del pipeline en Azure DevOps, los resultados de ejecución por módulo, y los gráficos obtenidos a partir de la encuesta aplicada al jefe del área de DevOps, quien participó activamente en la validación del proyecto. La inclusión de estos anexos tiene como objetivo brindar transparencia y trazabilidad al proceso, permitiendo que los lectores, jurados o revisores académicos puedan contrastar los resultados obtenidos con las fuentes originales y verificar la coherencia entre el diseño teórico y la práctica implementada. De manera general, los anexos se organizan así:

- Anexo A: Encuesta aplicada al jefe de DevOps (formulario completo y respuestas).


**Figura 6.1:** Entrevista jefe DevOps - Pregunta 1. Elaboración propia


**Figura 6.2:** Entrevista jefe DevOps - Pregunta 2 y 3. Elaboración propia

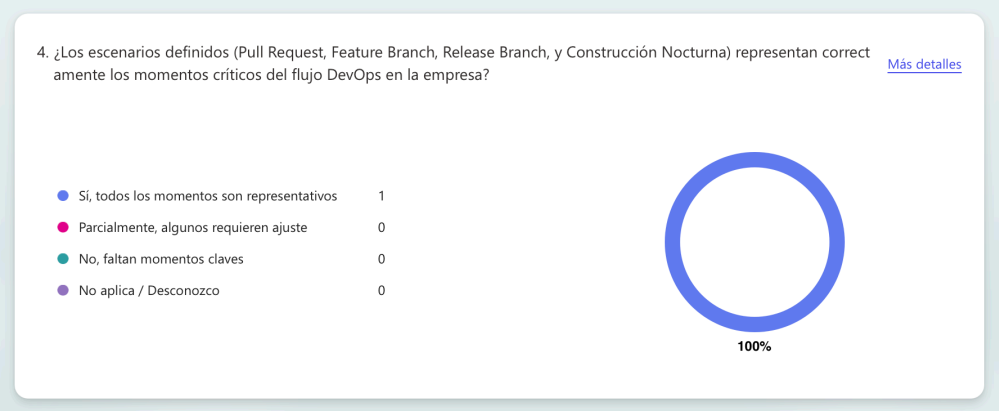
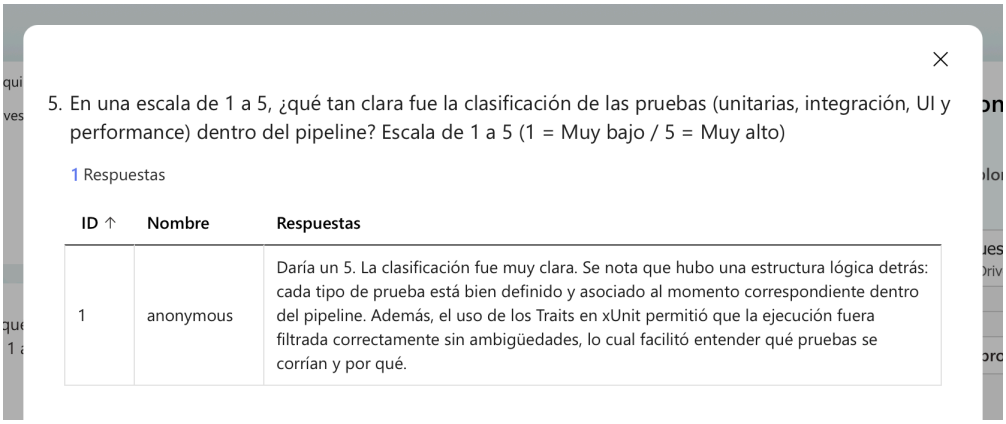



**Figura 6.3:** Entrevista jefe DevOps - Pregunta 4. Elaboración propia



5. En una escala de 1 a 5, ¿qué tan clara fue la clasificación de las pruebas (unitarias, integración, UI y performance) dentro del pipeline? Escala de 1 a 5 (1 = Muy bajo / 5 = Muy alto)

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Daría un 5. La clasificación fue muy clara. Se nota que hubo una estructura lógica detrás: cada tipo de prueba está bien definido y asociado al momento correspondiente dentro del pipeline. Además, el uso de los Traits en xUnit permitió que la ejecución fuera filtrada correctamente sin ambigüedades, lo cual facilitó entender qué pruebas se corrían y por qué. |

**Figura 6.4:** Entrevista jefe DevOps - Pregunta 5. Elaboración propia


**Figura 6.5:** Entrevista jefe DevOps - Pregunta 6 y 7. Elaboración propia


**Figura 6.6:** Entrevista jefe DevOps - Pregunta 8 y 9. Elaboración propia


**Figura 6.7:** Entrevista jefe DevOps - Pregunta 10 y 11. Elaboración propia

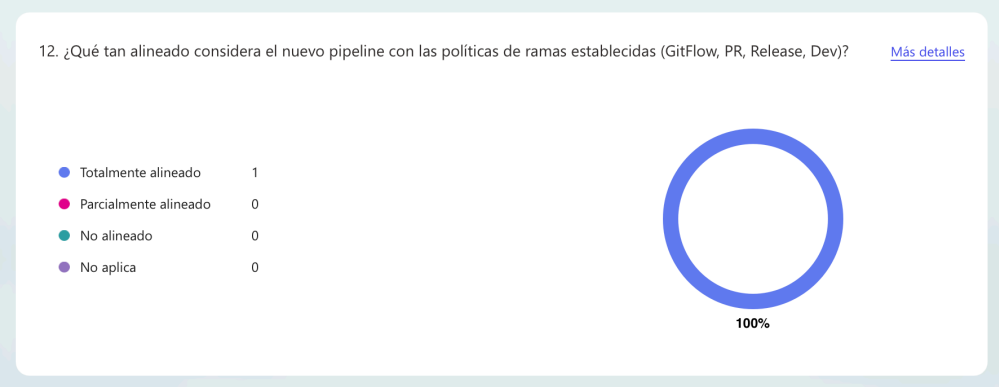



**Figura 6.8:** Entrevista jefe DevOps - Pregunta 12. Elaboración propia

13. Califique la estabilidad del pipeline luego de la implementación (1 = Muy inestable / 5 = Muy estable) Escala de 1 a 5 (1 = Muy bajo / 5 = Muy alto)

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Daría un 5. El pipeline se mostró estable incluso después de varios ciclos de ejecución en entornos reales. No se presentaron fallos inesperados en los jobs condicionales ni errores por conflictos de versiones o dependencias. La integración con Azure DevOps respondió bien y los tiempos de ejecución se mantuvieron constantes, lo que refleja una configuración sólida y madura. |

**Figura 6.9:** Entrevista jefe DevOps - Pregunta 13. Elaboración propia

14. ¿Qué aspectos del nuevo proceso destacaría como sus principales fortalezas?

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Destacaría principalmente tres cosas: La reducción del tiempo de retroalimentación: Antes las pruebas completas demoraban demasiado; ahora, solo se ejecutan las necesarias según el módulo afectado, lo que optimizó el tiempo en cada Pull Request y la productividad. La claridad y automatización: La matriz de momentos vs. tipos de prueba deja todo muy claro. Ya no depende del criterio individual del desarrollador, sino que el sistema decide automáticamente qué correr. La escalabilidad: El enfoque modular permite incorporar nuevos tipos de pruebas sin alterar la estructura general del pipeline, lo que le da flexibilidad a largo plazo. |

**Figura 6.10:** Entrevista jefe DevOps - Pregunta 14. Elaboración propia

15. ¿Qué limitaciones o posibles riesgos identifica en el proceso implementado?

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Una limitación potencial está en la detección de cambios. En proyectos más grandes o con arquitecturas distribuidas puede ser necesario un mecanismo más avanzado para identificar dependencias indirectas. También, aunque el proceso reduce tiempo, requiere disciplina en el uso de etiquetas y convenciones dentro de los tests. Si el equipo no sigue la misma nomenclatura de Traits o no actualiza las políticas de ramas, podría romperse la consistencia del sistema. |

**Figura 6.11:** Entrevista jefe DevOps - Pregunta 15. Elaboración propia

16. ¿Qué sugerencias haría para futuras mejoras o escalabilidad del pipeline a otros proyectos?

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Recomendaría integrar un módulo de métricas más visual, por ejemplo, dashboards automáticos con los tiempos de ejecución por tipo de prueba y tasa de fallos. También sería interesante extender el pipeline hacia pruebas de carga o rendimiento ejecutadas bajo demanda, y vincular la lógica de detect-changes con los repositorios secundarios en caso de soluciones multi-repo. Por último, incluir validaciones de cobertura de código en cada etapa permitiría tener una visión más completa del impacto de los cambios. |

**Figura 6.12:** Entrevista jefe DevOps - Pregunta 16. Elaboración propia

17. En general, ¿cómo calificaría la efectividad del nuevo proceso de integración continua propuesto?

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | El proceso es altamente efectivo. Cumple con su propósito de optimizar tiempos, mantener calidad y entregar retroalimentación oportuna sin comprometer estabilidad. Además, generó un cambio positivo en la forma en que los desarrolladores perciben las pruebas: ahora las ven como una ayuda, no como un obstáculo. |

**Figura 6.13:** Entrevista jefe DevOps - Pregunta 17. Elaboración propia


**Figura 6.14:** Entrevista jefe DevOps - Pregunta 18 y 19. Elaboración propia

20. ¿Desea agregar algún comentario o reflexión sobre el impacto del proyecto en la cultura DevOps o en el flujo de trabajo del equipo?

1 Respuestas

| ID ↑ | Nombre | Respuestas |
|---|---|---|
| 1 | anonymous | Sí. Creo que este proyecto ayudó a fortalecer la cultura DevOps en la empresa, porque demostró que automatizar con propósito sí mejora la productividad. Antes había cierta resistencia a ejecutar pipelines completos por el tiempo que tomaban; ahora el equipo confía más en el proceso, entiende su valor y lo usa con más frecuencia. También fomentó conversaciones más técnicas entre QA y desarrollo, lo que antes era poco común. En general, marcó un cambio tangible en la forma en que el equipo entiende la integración continua como parte del ciclo de desarrollo, no como una tarea aislada. |

**Figura 6.15:** Entrevista jefe DevOps - Pregunta 20. Elaboración propia

- Anexo B: Evidencias gráficas de configuración y ejecución de los pipelines (capturas de Azure DevOps).

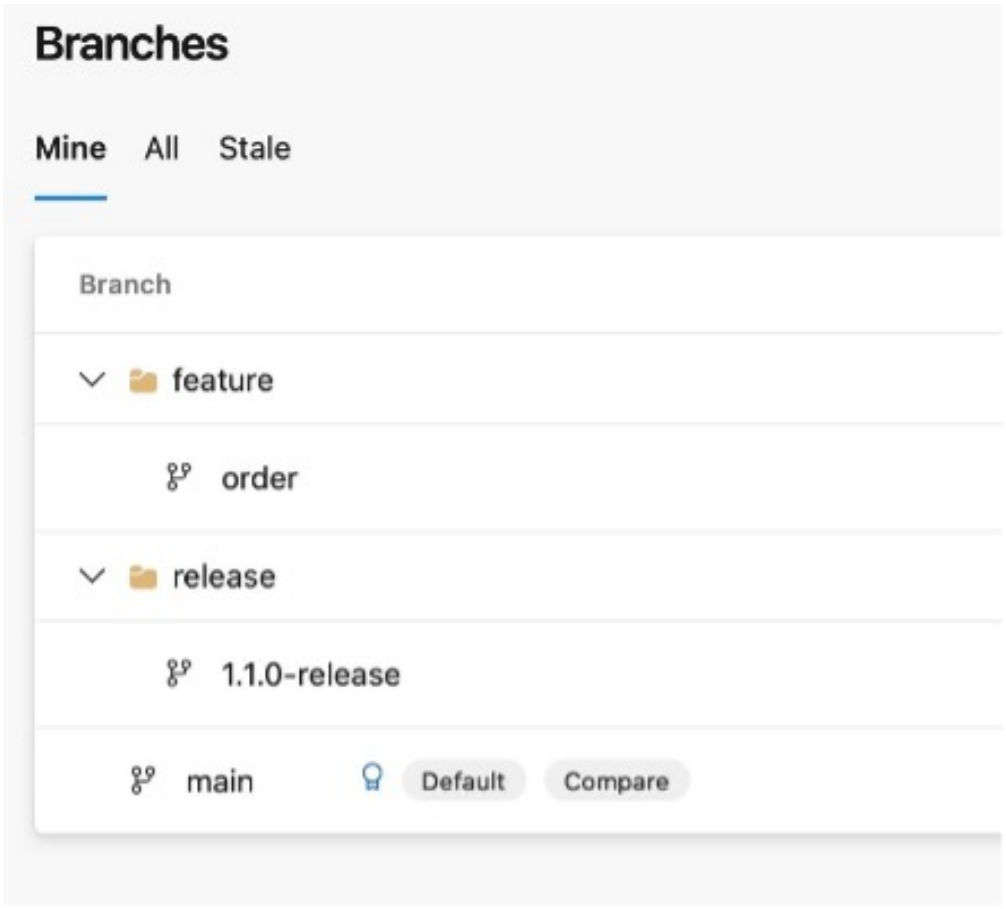



**Figura 6.16:** Creación de ramas del repositorio. Elaboración propia

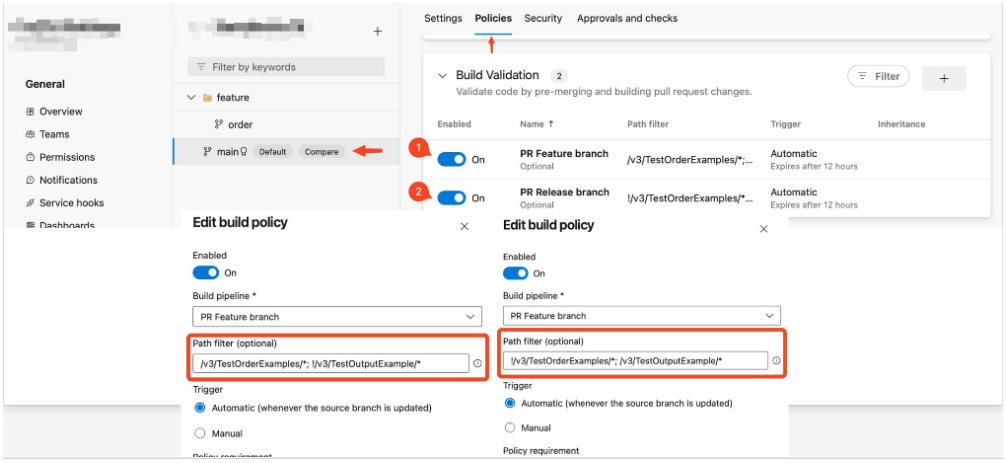



**Figura 6.17:** Configuración de politicas sobre las ramas. Elaboración propia

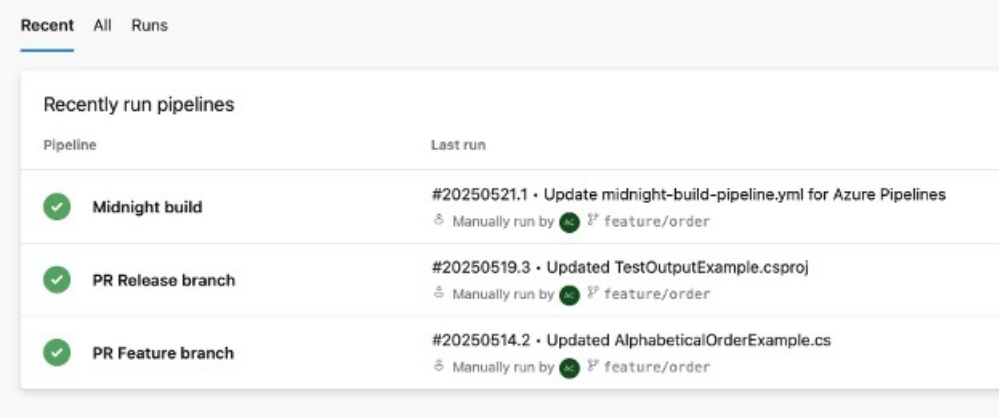



**Figura 6.18:** Resultado de ejecución de cada uno de los pipelines. Elaboración propia

- Anexo C: Matriz de momentos vs. tipos de prueba aplicada en el entorno de la empresa.

**Tabla 6.1:** Matriz visual de momentos vs. tipos de prueba

| Cadencia (CI-ViT) | Momento en el pipeline | Objetivo de feedback | Unitaria | Integración | E2E |
|---|---|---|---|---|---|
| **Commit / PR** | PR — Feature branch | Validar únicamente el módulo afectado y detectar fallas rápidas antes del *merge*. | ✓ | – | – |
| **Por cambio significativo** | PR — Release branch | Verificar interacciones entre módulos y estabilidad general del *release*. | ✓ | ✓ | – |
| **Diaria** | Build nocturno (CRON) | Detectar regresiones globales y evaluar la salud completa del repositorio. | ✓ | ✓ | ✓ |

La Tabla 6.1 resume de forma visual la relación entre los momentos del flujo de integración continua y los tipos de prueba ejecutados en cada etapa.

# Bibliografía


Anonymous (2023). Continuous testing is the key to ensuring quality in the devops era. *devops.com*. Accessed 2025-08-30.

AppVeyor Systems Inc. (2014). Appveyor ci. https://www.appveyor.com/. Servicio de CI para Windows/.NET en la nube.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2 edition. Clásico que populariza CI dentro de XP. (La 2.ª edición es 2004; mantengo la clave solicitada).

Booch, G. (1990). *Object-Oriented Design with Applications*. Benjamin/Cummings.

Bosch, J. (2007). Towards testing after deployment. *Towards Testing After Deployment*.

Bosch, J. (2022). Boost your digitalization with continuous experimentation.

Bosch, J., Muccini, H., Clements, P., Lago, P., Crnkovic, I., and Kuzniarz, L. (2016). Architecting in the digital age. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 1–2. IEEE. Cita secundaria que menciona el uso de CIViT en práctica.

Circle Internet Services, Inc. (2014). Circleci platform. https://circleci.com/. Servicio de CI hospedado; plataforma ampliamente usada.

Cohn, M. (2009a). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.

Cohn, M. (2009b). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.

Cusumano, M. and Selby, R. (1995). *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. Free Press.

Downs, J., Patterson, C., and Viger, E. (2010). Continuous integration in the real world. *IEEE Software*, 27(4):18–25.

Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional.

Elsner, K., Pannach, L., Heinemann, F., Wiesner, D., Fischer, S., and Pretschner, A. (2022). Build system aware multi-language regression test selection in continuous integration. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 176–185. IEEE.

Fallahzadeh, E., Bavand, A. H., and Rigby, P. C. (2023). Accelerating continuous integration with parallel batch testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*, pages 55–67. ACM.

Fowler, M. (2006). *Continuous Integration*. ThoughtWorks.

GitHub (2019). Github actions general availability. https://github.blog/changelog/2019-11-13-github-actions-is-generally-available/. GA en nov. de 2019.

GitLab Inc. (2015). Gitlab ci/cd. https://about.gitlab.com/releases/2015/09/22/gitlab-8-0-released/. CI integrado a partir de GitLab 8.0 (2015).

Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2018). Usage, costs, and benefits of continuous integration in open-source projects. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437.

Jenkins Project (2011). Jenkins: An extendable open source ci server. https://www.jenkins.io/project/history/. El fork de Hudson que devino en Jenkins; primera liberación estable en 2011.

Kawaguchi, K. (2008). Hudson ci. https://wiki.eclipse.org/Hudson-ci. Servidor de CI precursor de Jenkins.

Kolawole, I. A. and Fakokunde, A. (2024). Improving software development with continuous integration and deployment for agile devops in engineering practices. *International Journal of Computer Applications Technology and Research (IJCATR)*, 14(1):25–39.

Li, Q., Zhou, H., and Wu, J. (2023). Accelerating continuous integration with parallel batch testing. *arXiv preprint*. Accessed: 2025-08-21.

Machalica, M., Samylkin, A., Porth, M., and Chandra, S. (2019). Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE.

Mehta, S., Farmahinifarahani, F., Bhagwan, R., Gupta, S., Jafari, S., Kumar, R., Saini, V., and Santhiar, A. (2021a). Data-driven test selection at scale. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, pages 1766–1777. ACM.

Mehta, S., Farmahinifarahani, F., Bhagwan, R., Gupta, S., Jafari, S., Kumar, R., Saini, V., and Santhiar, A. (2021b). Data-driven test selection at scale. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1766–1777. ACM.

Mens, T., Adams, B., et al. (2023). Empirical analysis of github actions workflows: Adoption and usage trends. *Empirical Software Engineering*.

Microsoft Learn (2024). Configurar ejecuciones programadas (schedules) en azure pipelines yaml. https://learn.microsoft.com/azure/devops/pipelines/process/scheduled-triggers. Consultado: 2025-10-14.

Microsoft Learn (2025). Configurar políticas de rama para validar pull requests. https://learn.microsoft.com/azure/devops/repos/git/branch-policies. Consultado: 2025-10-14.

Nilsson, P., Staron, M., and Nyholm, L. (2014). Civit: Continuous integration visualization technique. In *Proceedings of the Software Center Research*. Modelo de visualización de actividades de prueba en CI. Publicación referida en trabajos del Software Center; algunas versiones circulan como informe técnico.

Peño, J. (2015). *Técnicas de pruebas de software: estáticas y dinámicas*. PhD thesis, Universidad Politécnica de Madrid.

Piattini, M., Polo, M., and Ruiz, F. (2013). Advances in test automation for software quality. *Journal of Software: Evolution and Process*, 25(12):1221–1232.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

Ron Powell, M. S. (2019). *The Data-Driven Case for CI: What 30 Million Workflows Reveal About DevOps in Practice*. Circle CI.

Rothermel, G. (2018). The past, present, and future of regression test selection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA).

Schwendner, D., Jungwirth, M., Gruber, M., et al. (2025). Practical pipeline-aware regression test optimization for continuous integration. In *arXiv*. Accessed 2025-08-30.

ThoughtWorks Open Source (2001). Cruisecontrol. https://cruisecontrol.sourceforge.net/. Proyecto de CI de código abierto iniciado a comienzos de los 2000.

Travis CI (2011). Travis ci timeline. https://blog.travis-ci.com/2011-01-11-travis-ci-blast-off/. Lanzamiento y adopción inicial en proyectos OSS.

van Veenendaal, E. (2018). *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 4th edition.

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in github. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 805–816. Usado como respaldo de adopción de CI en repositorios Git; clave mantenida como Golzadeh2021 para compatibilidad con el .tex.

vom Brocke, J., Hevner, A. R., and Maedche, A. (2022). Introduction to design science research. In *Design Science Research. Cases*, pages 1–13. Springer. Accessed: 2025-08-26.

Wang, Y., Mäntylä, M., Liu, Z., and Markkula, J. (2022). Test automation maturity improves product quality – quantitative study of open source projects using continuous integration. In *arXiv*. Accessed 2025-08-30.